Chapter 3

Firmware Approach

1.1 Basics of Firmware

At one time, software developers who wanted to build applications to run under Windows had no choice but to write them in C. Such programs connected directly to the windows application programming interface (API), a bare bone set of functions provided by the OS. In those days, besides a Windows compatible computer, the only available development tools were packaged by Microsoft into the Software Development kit, or SDK.

Learning to use the SDK to develop Windows programs has always been a difficult task. One of the reasons for this is it requires a new way of thinking. Graphical output is quite different than character based output. It takes additional time and effort to figure out that one need to anticipate all different types of input that user will send to a program. Also, one must get use to working with all the quirks in the user interface objects – windows, menus, cursors, dialogues, icons and so on – the windows provides. These build in objects let the programming be productive once a person learns how to use them. Programmers can use C++ for building windows application. With its build-in C compatibility, C++ can be used for SDK style programming [89]. However, such use does not take advantage of the object oriented features of the C++ language. Realizing
the benefits of those features requires a class library that’s written to simplify the tedious housekeeping chores and frequent quirks that makes SDK programming so difficult. Several such libraries are commercially available, including Borland’s Object Windows Library (OWL) and Microsoft’s Microsoft Foundation Class (MFC) library. The MFC library is an attempt by Microsoft to provide C++ programmers with an object oriented interface to windows. Microsoft created the MFC library with the goal of facilitating and simplifying the process of programming for Microsoft Windows.

Thus, Visual C++ is a visual development tool that makes use of the MFC library to make the development under Windows environment faster and reliable.

MFC library is a collection of C++ classes. It is provided as a Dynamic Link Library (DLL) so any application has access to the classes in MFC. A DLL consist of executable functions that are loaded into memory and are independent from any application. Libraries such as MFC are called application frameworks, because they give the user a framework for an application. The MFC classes have been built using the OS’s API functions.

The first automated ECG programs were developed in the 1970s, when digital ECG machines became possible by third generation digital signal processing boards. Commercial models, such as those developed by Hewlett Packard incorporated these programs into clinically-used devices.

During the 1980s and 1990s, extensive research was carried out by companies and by university labs in order to improve the accuracy rate, which was not very large in the first models. For this purpose, several
signal databases with normal and abnormal ECGs were built up by institutions such as MIT and used to test the algorithms and its accuracy.

3.2 Phases of Holter recorder

1. A digital representation of each recorded ECG channel is obtained, by means of an analog-digital conversion device and a special data acquisition software or a digital signal processing (DSP) chip.

2. The resulting digital signal is processed by a series of specialized algorithms, which start by conditioning it, e.g., removal of noise, base-level variation, etc.

3. Feature extraction: mathematical analysis is now performed on the clean signal of all channels, to identify and measure a number of features which are important for interpretation and diagnosis, this will constitute the input to AI-based programs, such as the peak amplitude, area under the curve, displacement in relation to baseline, etc, of the P, Q, R, S and T waves, the time delay between these peaks and valleys, heart rate frequency (instantaneous and average), and many others. Some sort of secondary processing such as Fourier analysis and wavelet analysis may also be performed in order to provide input to pattern recognition-based programs.

4. Logical processing and pattern recognition, using rule-based expert systems, probabilistic Bayesian analysis or fuzzy logics algorithms, cluster analysis, artificial neural networks, genetic algorithms and others techniques are used to derive conclusions, interpretation and diagnosis

5. A reporting program is activated and produces a proper display of original and calculated data, as well as the results of automated interpretation.
6. In some applications, such as automatic defibrillators, an action of some sort may be triggered by results of the analysis, such as the occurrence of an atrial fibrillation or a cardiac arrest, the sounding of alarms in a medical monitor in intensive-care unit applications, and so on.

3.3 Analyzing Software of Holter Recorder

When the recording of ECG signal is finished (usually after 24 or 48 hours), it is up to the physician to perform the signal analysis. Since it would be extremely time demanding to browse through such a long signal, there is an integrated automatic analysis process in each Holter software which automatically determines different sorts of heart beats, rhythms, etc. However the success of the automatic analysis is very closely associated with the signal quality. The quality itself mainly depends on the attachment of the electrodes to the patient body. If these are not properly attached, the electromagnetic disturbance surrounding us will influence the ECG signal resulting thus in a very noisy record. If the patient moves rapidly, the distortion will be even bigger. Such record is then very difficult to process. Besides the attachment and quality of electrodes, there are another factors affecting the signal quality, such as muscle tremors, sampling rate and resolution of the digitalized signal (high quality devices offer 2000Hz and 16 bits or higher) [28].

The automatic analysis commonly provides the physician with information about heart beat morphology, beat interval measurement, heart rate variability, rhythm overview and patient diary (moments when the patient pressed the patient button). Advanced systems also perform the spectral analysis, ischemic burden evaluation, graph of patient’s activity or
PQ segment analysis. Another requirement is the ability of pacemaker detection and analysis. Such ability is useful when one wants to check the correct pacemaker function.
3.4 ECG Analysis System Software

3.4.1 Interpretation of ECG

Interpretation of 12 lead ECG using software or computers uses algorithms to determine whether a patient is normal or abnormal. It helps in determining the abnormalities if discovered. Right back from era of 1960s’ when some mainframe computers centrally located in computing centers performed the ECG analysis to the modern period ECG analysis machines that are minicomputer based; interpretation of the ECG signal has improved to a great extent. Today’s machines help perform the complete data acquisition, processing and interpretation of the ECG signal at the patient’s bedside without transmitting any data to remote computer [90].

The modern microprocessor based ECG interpretive machines include eight ECG amplifiers so that they can simultaneously sample and store eight leads. They then synthesize the other four redundant leads. These machines include enough memory to store all leads for a 10 second interval at a clinical sampling rate of 250sps – 500sps. ECG interpretation starts with feature extraction, which has two parts; 1) Waveform recognition to identify the waves in the ECG signal and, 2) measurement to quantify a set of amplitudes and time durations that is to be used to drive the decision making process. The decision is made based in comparison to the standard values that are fed in the computer on which it operates.
We propose a system that performs all the basic tasks involved in interpretation of the 12 lead ECG signal and at the same time performs various other functions which involves data management from previous cases, maintaining log, transmission of information, etc.

The analysis system defined performs complete 12-lead ECG interpretation. The computer approaches the ECG interpretation as a pattern classification problem and applies a multivariate statistical pattern recognition method to solve it. The system mimics the human expert’s decision process using a rule based expert system. During the ECG analysis the system performs various mathematical tasks to extract and calculate the different features of the ECG signal. It calculates the P wave and T wave interval, detects QRS complex and measures its interval. The system determines the RR peak interval. It looks out for any missing or abnormal waveforms [5]. Finally it predicates the most possible arrhythmia from the derived parameters using the predefined condition. In our development, we gave importance in designing a system to analyze ECG waveform. The analyzing of the signal helps in understanding various patterns of ECG signal and execute appropriate decision about the health of the patient. This system deals with implementation of algorithms like detrending of ECG signal, RR interval detection, QRS complex detection, detection of pre ventricular contractions and other abnormalities using embedded controllers. The main component of the firmware includes;

1. Software using High level language C++ to analyze it
2. GUI designed using Visual basic
As the main aim was to develop an embedded system, algorithms for determination of RR interval, detrending of RR waveform, QRS detection, and Beat Rate detection were designed and implemented in higher level language. The higher level languages used in the applications are C++ and Visual Basics. C++ is used for calculative and numerical reasoning including decision making whereas Visual basics is used for designing HMI and displaying the main result and waveform on the console.
3.5 System Flow Algorithm

ECG analysis system is designed to function as follows:

Whenever it is started it asks whether the patient is new or old.

NOW,

1 If the patient is new, then the system gives privilege to fill details of the new patient and register it. This is done so that his/her ECG waveform will be directly stored in his/her record which can be accessed for future reference. Every patient is given an own unique numerical ID.

2 If the patient was previously registered then we can select the register option to check for his records present in the system.

3 Once the patient is registered, the system checks the serial ports to capture the ECG waveform received by the hardware and display it on the screen.

4 Before displaying the data, the system performs some set of operations in order to remove noise, smoothens the signal, etc.

5 Once the data is displayed, simultaneously, it analyzes the acquired waveform to display heart rate.

6 It calculates heart rate, R wave width or signal width, detects QRS complex.
7 Using the analyzed data, ECG analysis system checks through the algorithm for a probable arrhythmia. This is important and unique feature available in the system to display the type of arrhythmia which will helps the doctor to take the fast remedial action and the derived arrhythmia is displayed on the screen.

8 The output of ECG analysis system i.e. the status of the patient Normal/abnormal along with types of arrhythmia can then be messaged using GSM module. This is a unique feature available in the analysis system. Transmission of ECG analysis output is very important.
START

New Patient ?

Enter Details of the Patient

Start Capturing ECG

Eradicate Noise & Store the Data

Review Test from existing Data

Display the Waveform on Screen

Analyze the Data using Visual C++

Show analyzed result

Transmit Data?

Print Result?

Review Test with report printing

Enter destination details

STOP

Figure 3.1: System Flow Algorithm
3.6 Computing [Signal Processing]

Computing is usually defined as the activity of using and improving computer technology, computer hardware and software. It is the computer-specific part of information technology. Computer science (or computing science) is the study and the science of the theoretical foundations of information and computation and their implementation and application in computer systems.

Computing Curricula 2005 defined computing: [37]

In a general way, we can define computing to mean any goal-oriented activity requiring, benefiting from, or creating computers. Thus, computing includes designing and building hardware and software systems for a wide range of purposes; processing, structuring, and managing various kinds of information; doing scientific studies using computers; making computer systems behave intelligently; creating and using communications and entertainment media; finding and gathering information relevant to any particular purpose, and so on. The list is virtually endless, and the possibilities are vast.

A computer is a machine that manipulates data according to a set of instructions called a computer program. The program has an executable form that the computer can use directly to execute the instructions. The same program in its human-readable source code form, enables a programmer to study and develop the algorithm. Because the instructions can be carried out in different types of computers, a single set of source
instructions converts to machine instructions according to the central processing unit type.

The execution process carries out the instructions in a computer program. Instructions express the computations performed by the computer. They trigger sequences of simple actions on the executing machine. Those actions produce effects according to the semantics of the instructions.

Computer programming in general is the process of writing, testing, debugging, and maintaining the source code and documentation of computer programs. This source code is written in a programming language, which is an artificial language, restrictive, demanding, and unforgiving to humans but easily translated by the computer. The purpose of programming is to invoke the desired behavior (customization) from the machine. The process of writing high quality source code requires knowledge of both the application's domain and the computer science domain. The highest quality software is thus developed by a team of various domain experts, each person a specialist in some area of development. But the term programmer may apply to a range of program quality, from hacker to open source contributor to professional [89]. And a single programmer could do most or all of the computer programming needed to generate the proof of concept to launch a new "killer" application.
### 3.6.1 Programming languages

Different programming languages support different styles of programming (called *programming paradigms*). The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference. Ideally, the programming language best suited for the task at hand will be selected. Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of *compilers* for that language, and the efficiency with which programs written in a given language execute.

The details look different in different languages, but a few basic instructions appear in just about every language:

- **input**: Get data from the keyboard, a file, or some other device.
- **output**: Display data on the screen or send data to a file or other device.
- **arithmetic**: Perform basic arithmetical operations like addition and multiplication.
- **conditional execution**: Check for certain conditions and execute the appropriate sequence of statements.
- **repetition**: Perform some action repeatedly, usually with some variation.

Many computer languages provide a mechanism to call functions provided by libraries. Provided the functions in a library follow the appropriate run time conventions (e.g., method of passing arguments), then these functions may be written in any other language.
3.6.2 Modern programming

Whatever the approach to software development may be, the final program must satisfy some fundamental properties. The following properties are among the most relevant:

- **Efficiency/performance**: the amount of system resources a program consumes (processor time, memory space, slow devices such as disks, network bandwidth and to some extent even user interaction): the less, the better. This also includes correct disposal of some resources, such as cleaning up temporary files and lack of memory leaks.

- **Reliability**: how often the results of a program are correct. This depends on conceptual correctness of algorithms, and minimization of programming mistakes, such as mistakes in resource management (e.g., buffer overflows and race conditions) and logic errors (such as division by zero).

- **Robustness**: how well a program anticipates problems not due to programmer error. This includes situations such as incorrect, inappropriate or corrupt data, unavailability of needed resources such as memory, operating system services and network connections, and user error.

- **Usability**: the ergonomics of a program: the ease with which a person can use the program for its intended purpose, or in some cases even unanticipated purposes. Such issues can make or break its success even regardless of other issues. This involves a wide range of textual, graphical and sometimes hardware elements that improve the clarity, intuitiveness, cohesiveness and completeness of a program's user interface.

- **Portability**: the range of computer hardware and operating system platforms on which the source code of a program can be
compiled/interpreted and run. This depends on differences in the programming facilities provided by the different platforms, including hardware and operating system resources, expected behavior of the hardware and operating system, and availability of platform specific compilers (and sometimes libraries) for the language of the source code.

- **Maintainability**: the ease with which a program can be modified by its present or future developers in order to make improvements or customizations, fix bugs and security holes, or adapt it to new environments. Good practices during initial development make the difference in this regard. This quality may not be directly apparent to the end user but it can significantly affect the fate of a program over the long term.

### 3.6.3 Algorithmic complexity

The academic field and the engineering practice of computer programming are both largely concerned with discovering and implementing the most efficient algorithms for a given class of problem. For this purpose, algorithms are classified into orders using so-called **Big O notation**, $O(n)$, which expresses resource use, such as execution time or memory consumption, in terms of the size of an input. Expert programmers are familiar with a variety of well-established algorithms and their respective complexities and use this knowledge to choose algorithms that are best suited to the circumstances.
3.7 Software Implemented

3.7.1 Determination of RR interval

ECG was recorded continuously, during a passive event related potential paradigm, where subject sat in a chair while auditory pitch stimuli were delivered to right ear. Discrete event series, \( R_i = R_i - 1 \) intervals as a function of \( R_i \) occurrence times, was constructed by an adaptive QRS detector algorithm. As a result of the detection algorithm an unevenly sampled RR interval time series was obtained. In order to recover an evenly sampled signal from the irregularly sampled event series cubic interpolation was applied. Sampling rate of the ECG was 350 Hz. Discrete event series, \( R_i - R_{i-1} \) intervals as a function of \( R_i \) occurrence times, was constructed by an adaptive QRS detector algorithm. As a result of the detection algorithm an unevenly sampled RR interval time series was obtained. In order to recover an evenly sampled signal from the irregularly sampled event series cubic interpolation was applied. \[41\]

We denote the RR interval time series as

\[
Z = (R2 - R1; R3 - R2; \ldots; RN - RN-1)^T \in \mathbb{R}^{N-1}
\]

where \( N \) is the number of R peaks detected. The RR series can be considered to consist of two components

\[
Z = Z_{\text{stat}} + Z_{\text{trend}}
\]
where $z_{\text{stat}}$ is the nearly stationary RR series of interest and $z_{\text{trend}}$ is the low frequency aperiodic trend component. The trend component can be modeled with a linear observation model as

$$z_{\text{trend}} = H\theta + \nu$$

where $H \in \mathbb{R}^{(N-1)\times M}$ is the observation matrix, $\theta \in \mathbb{R}^M$ are the regression parameters and $\nu$ is the observation error. The task is then to estimate the parameters by some fitting procedure so that the prediction $\hat{z}_{\text{trend}} = H\hat{\theta}$ be used as the estimate of the trend. The properties of the estimate depend strongly on the properties of the basis vectors (columns of the matrix $H$) in the fitting. Widely used method for the solution of the estimate $\hat{\theta}$ is the least squares method.

**Program implemented:**

// DetectR.cpp: implementation of the DetectR class.
#include "stdafx.h"
#include "StPlot.h"
#include "DetectR.h"
#include "math.h"
#ifdef _DEBUG
#undef THIS_FILE
#undef DEBUG_NEW
#endif

float max_array(float wkept[], int len);
float min_array(float min_1, float min_2);
int value = 0, step = 0;
int R_count = 0;
int cn_t;
int r_off;
static xy;
int outputInt = 0;
#define scale 3
#define xmax 1.0039
#define dx 0.0039
#define prod_ScaleDx (int)(scale * dx)
#define prod_ScaleXMax (int)(scale * xmax)
#define Rate 100
#define tra 4
#define data_length 400
#define Resultant_length (int)(400 + prod_ScaleXMax)
float K[prod_ScaleXMax];
float fun_1[Resultant_length];
float Diff[Resultant_length];
float Conv[Resultant_length];
float wkept[Resultant_length];
int timecounter = 0, vpb_per_min = 0;
int abnormal_count = 0, normal_count = 0, set_detect_flag = 0;
int vpb_pointer = 0;

DetectR::DetectR()
{
}
DetectR::~DetectR()
{
}

short DetectR::WaveRDetect(double * inputSignal, double * outputX_Array, double * outputY_Array)
{
    R_count = 0;
    cn_t = 0;
    for(int i = 0; i < 1200; i++)
    {
        inputInt[i] = (int)*(inputSignal + i);
    }
    DetectRWave(inputInt, outputIntX, outputIntY);
    for(i = 0; i <= R_count; i++)
    {
        *(outputX_Array + i) = (double)outputIntX[i];
        *(outputY_Array + i) = (double)outputIntY[i];
    }
    return R_count;
}

void DetectR::ResetMe()
{
    for(int i = 0; i < prod_ScaleXMax; i++)
    {
    }
K[i] = 0;
}

for(i = 0; i < Resultant_length; i++)
{
        Conv[i] = 0;
}

for(i = 0; i < 1200; i++)
{
        outputIntX[i]= 0;
        outputIntY[i]= 0;
}

R_count = 0;
if (timecounter == 15)
{
        timecounter = 0;
        vpb_per_min = 0;
}
}

short DetectR::WaveDetect(short * input, short * outputX, short * outputY)
{
        R_count = 0;
        cn_t = 0;
int Q_value = 0, S_value = 0, VPB_value = 0, k = 0;
int sampling = 3, op_width = 0;
for(int i = 0; i < 1200; i++)
{
    if(sampling == 3)
    {
        sampling = 0;
        inputInt[k] = (int)(*(input + i)/5);
        k++;
    }
    sampling++;
}
DetectRWave(inputInt, outputIntX, outputIntY);
for(i = 0; i <= abs(R_count); i++)
{
    *(outputX + i) = (int)outputIntX[i];
    *(outputY + i) = (int)outputIntY[i];
    op_width = FindQRS((int)outputIntX[i], Q_value, S_value);
    *(outputY + i) = op_width;
}
timecounter ++;
return (R_count + 1);
}
void DetectR::DetectRWave(int *signal, int *x_location, int *y_location)
{
    int i = 0, j = 0, k = 0, first = 0, scnd = 0, count = 0;
    unsigned char check = 0;
int x_pos[100], x_lock[30];
float least_value = -1000.0;
int timer, final = 0, high_value = 2000, low_value = -2000, final_value = -3200, qrs_final = 0;
int greater_value = 1000;
float multi, optim_ze = 0;

fun_1[0] = 0;
fun_1[1] = 0.3360;
fun_1[2] = 0.3320;
fun_1[3] = 0;

for (i = 0; i < (Resultant_length); i++)
{
    if (i >= data_length)
        signal[i] = 0;

    if (i >= prod_ScaleXMax)
        fun_1[i] = 0;
}

multi = (float)sqrt(scale);
for (k = 0; k < 400; k++)
{
    if (k <= 1)
    {

    
}
optim_ze = (*(signal + j)) * (*(fun_1 + k - j)));
 Conv[k] = Conv[k] + optim_ze;

else
{
    for( j = k - 2; j < k; j++)
    {
        optim_ze = (*(signal + j)) * (*(fun_1 + k - j));
        Conv[k] = Conv[k] + optim_ze;
    }
}

if(k > 0 )
{
    Conv[k - 1] = -((Conv[k - 1])-(Conv[k])); // * multi;
    if(Conv[k - 1] != 0)
    {
        Conv[k - 1] = Conv[k - 1] * multi;
        if (abs(Conv[k - 1]) > least_value)
            least_value = abs(Conv[k - 1]);
    }
}

j = 0;
for (first = 10;first < data_length;first++)
{
    if (abs(Conv[first]) > least_value * 0.35)
{  
    x_pos[count] = first;
    count++;
}

// sorting the real R points from the above points.
for (scnd = 0; scnd <= count - 1; scnd++)
{  
    if (x_pos[scnd + 1] - x_pos[scnd] > 20)
    {
        x_lock[check] = x_pos[scnd];
        x_lock[check + 1] = x_pos[scnd + 1];
        check++;
    }
}

final = 0;
if (check == 0)
return;
for (scnd = 0; scnd <= check; scnd++)
{
    for (timer = x_lock[scnd] - (int)(Rate * 0.2); timer <
        x_lock[scnd] + (int)(Rate * 0.03); timer++)
    {
        (timer < 0 || timer >= data_length - 1)
        goto label;
        if (abs(signal[timer]) >= final_value)
{ 
    x_location[final] = timer;
    y_location[final] = signal[timer];
    final_value = abs(signal[timer]);

    label:;
}

final++;
final_value = -3200;

if (final == 0)
    R_count = final;
else
    R_count = final - 1;

int DetectR::FindBeatStart(int Rpoint)
{
    int Mult;
    int i, slope_1, slope_2, amplitude, FindStartOfWave;

    if(Rpoint < (int)(Rate * 0.03))
        Rpoint = (int)(Rate * 0.03);
    amplitude = abs(inputInt[Rpoint]);
    for (i = Rpoint - (int)(Rate * 0.03); i > Rpoint - (int)(Rate * 0.1); i--)
    {
        slope_1 = inputInt[i + 2] - inputInt[i + 1];
slope_2 = inputInt[i + 1] - inputInt[i];
Mult = slope_1 * slope_2;
if (Mult <= 0)
{
    FindStartOfWave = i + 1;
    break;
}
FindStartOfWave = i + 1;
return(FindStartOfWave);
}
int DetectR::FindBeatEnd(int Rpoint)
{
    int i,FindEndOfWave,slope_1,slope_2;
    int Mult;
    int amplitude;
    amplitude = abs(inputInt[Rpoint]);
    for (i = Rpoint ; i < Rpoint + (int)(Rate * 0.10);i++)
    {
        if (i >= Rate * tra)
            break;
        else
        {
            slope_1 = inputInt[i + 2] - inputInt[i + 1];
slope_2 = inputInt[i + 1] - inputInt[i];
Mult = slope_1 * slope_2;
            if (Mult <= 0)
FindEndOfWave = i + 1;
break;
}
}
FindEndOfWave = i + 1;
return(FindEndOfWave);
}
int DetectR::FindQRS(int R_value,int Q_x,int S_x)
{
    int width= 0;
    Q_x = FindBeatStart(R_value);
    S_x = FindBeatEnd(R_value);
    width = S_x - Q_x;
    VPB_Analysis(width);
    return(width);
}
void DetectR::VPB_Analysis(int QS_width)
{
    if (QS_width > Rate * 0.1 && QS_width < Rate * 3)
    {
        if (abnormal_count > 0 && normal_count > 0)
            set_detect_flag = 1;
        else
            abnormal_count = abnormal_count + 1;
    }
if (abnormal_count > 14)
    set_detect_flag = 1;

vpb_per_min = vpb_per_min + 1;
}

else
{
    if (abnormal_count > 0)
        normal_count = normal_count + 1;

    if (abnormal_count > 1 && normal_count > 1)
        set_detect_flag = 1;
}

if (set_detect_flag == 1)
{
    if (abnormal_count == 1 && normal_count == 1)
        vpb_pointer = 1; //"Bigeminy"
    else if (abnormal_count == 1 && normal_count == 2)
        vpb_pointer = 2; //"Trigeminy"
    else if (abnormal_count == 2 && normal_count > 1)
        vpb_pointer = 3; //"Pair PVC"
    else if (abnormal_count == 5 && normal_count > 1)
        vpb_pointer = 4; //"Run 5 PVC"
    else if (abnormal_count == 11 && normal_count > 1)
        vpb_pointer = 5; //"Run 11 PVC"
else if (abnormal_count > 14)
    vpb_pointer = 6; //" V.Tach"
    abnormal_count = 0;
normal_count = 0;
set_detect_flag = 0;
}
if (timecounter == 14)
{
    if (vpb_per_min == 24)
        vpb_pointer = 7;
    else if (vpb_per_min == 12)
        vpb_pointer = 8;
    else if (vpb_per_min == 6)
        vpb_pointer = 9;
}

3.7.2 PSD Estimation

Power spectral density (PSD) analysis provides the basic information of how power (variance) distributes as a function of frequency. Independent of the method used, only an estimate of the true PSD of the signal can be obtained from proper mathematical algorithms. [48]

Methods for PSD estimation can be classified as nonparametric (e.g. methods based on FFT) and parametric (methods based on autoregressive
(AR) time series modelling). In the latter approach the RR time series is modelled as an AR (p) process.

\[ Z_t = - \sum_{j=1}^{p} a_j Z_{t-j} + e_t, \quad t = p+1, \ldots, N-1 \quad 1 \leq j \leq p \]

where \( p \) is the model order, \( a_j \) are the AR coefficients and \( e_t \) is the noise term. A modified covariance method is used to solve the AR model. The power spectrum estimate \( P_z \) is then calculated as:

\[ P_z(\omega) = \sigma^2 / \left[ |1 + \sum \alpha_j e^{-i\omega j}|^2 \right] \]

### 3.7.3 Welch’s Method

Welch’s method (also called the periodogram method) for estimating power spectra is carried out by dividing the time signal into successive blocks, forming the periodogram for each block, and averaging. Denote the \( m \)th windowed, zero-padded frame from the signal \( x \) by

\[ x_m(n) = w(n)x(n + mR), \quad n = 0, 1, \ldots, M-1; \quad m = 0, 1, \ldots, K-1 \]

Where \( R \) is defined as the window hop size, and let \( K \) denote the number of available frames. The Welch estimate of the power spectral density is given by

\[ S_x^W(\omega_k) = \frac{1}{K} \sum_{m=0}^{K-1} P_{x_m,M}(\omega_k). \]

In frequency-domain the PSD is analyzed by calculating powers and peak frequencies for different frequency bands. The commonly used frequency
bands are very low frequency (VLF, 0-0.04 Hz), low frequency (LF, 0.04-0.15 Hz), and high frequency (HF, 0.15-0.4 Hz).

### 3.7.4 QRS Detection

The precision in the identification of QRS complexes is of great importance for the reliability of an automated ECG analyzing system and thus, for the diagnosis of cardiac diseases. [24]

The proposed method can be divided into three steps: 1) estimation of the initial R-wave fiducial points, 2) extraction of R-wave data points and modeling of the R-wave shape, and 3) correction of the fiducial point values using the estimated model. The second and last steps can be repeated until the fiducial point values converge. An illustrative diagram of the proposed method is presented in figure above and the three steps are described in details in the following.

Stepwise representation of QRS detection on algorithm;
Step 1: Estimation of initial R-Wave Fiducial points

At First, an adaptive QRS detector algorithm based on the one presented in is applied to detect the R-wave fiducial points from the sparsely sampled ECG recording. The accuracy of the observed time instants is then improved by QRS interpolation (a piecewise cubic spline interpolation with a sampling rate equal to 20 kHz). The observed R-wave maximums after interpolation are taken as initial guesses for R-wave fiducial points.

Step 2: Modeling the R-wave shape
Once the initial R-wave fiducial points have been estimated the shape of R-wave is modeled. For this, data points from each R-wave are extracted by using a time window centered at the corresponding initial fiducial point values. The data points of each R-wave are then accumulated into analogous temporal frame. The analogous temporal frame is obtained by subtracting the initial fiducial point instants from the time indices of the corresponding R-wave data points.

**Step 3: Correction of the R-wave fiducial point values**

The final step of the proposed method is to use the estimated R-wave model for correcting the initial fiducial point time instants. This is accomplished through linear LS regression. As regressors we select the estimated model $h(\hat{\theta}, t)$ and its first derivative $dh(\hat{\theta}; t)/dt$. The derivative is included in the regression to enable the shifting of the peak position in time.

### 3.8 Heart Rate Calculation

**Heart rate** is the number of heartbeats per unit of time - typically expressed as beats per minute (bpm) - which can vary as the body's need for oxygen changes, such as during exercise or sleep [79]. The measurement of heart rate is used by medical professionals to assist in the diagnosis and tracking of medical conditions. It is also used by individuals, such as athletes, who are interested in monitoring their heart rate to gain
maximum efficiency from their training. The **R wave to R wave interval** (RR interval) is the inverse of the heart rate.

Heart rate is measured by finding the pulse of the body. This pulse rate can be measured at any point on the body where an artery's pulsation is transmitted to the surface - often as it is compressed against an underlying structure like bone - by pressuring it with the index and middle finger. The thumb should not be used for measuring another person's heart rate, as its strong pulse may interfere with discriminating the site of pulsation. [80]

### 3.9 Detection of Abnormal Waveforms

Detection of abnormal waveform and missing waveform in the ECG signal is of great importance. Abnormal waveforms such as Pre-Ventricular Contraction (PVC), if present, can lead to error in the analysis. To detect these abnormal waveforms, we detect the QRS complexes in the ECG signal and then decompose it into set of frequency bands using wavelet transformation. To distinguish between “true” R-waves and PVCs, an adaptive threshold is implemented with a value greater than that of R-waves and less than the value of PVCs. After identifying the PVCs, they are eliminated with other aberrations in the signal in order to produce an R-wave.

**Program Implementation**

// Abnormal Waveform
short points_to_add, average_offset, average_initialised,
number_of_peaks;
short possible_R, clamp_cnt, max_data_point, R_peak_found;
short sec_6_cnt, average_R_R;
void DetectHR(double, short); // detect VPB
void DetectHR(int,int,int, int); // detect only HR
//void DetectHR(int);        // detect only HR
//void DetectHR(int,int,int,int);
short CalculateHRForLead2(int);
short CalculateHRForLeadV2(int);
short CalculateHRForLeadV5(int);
void CheckDetectionLead();
void CheckDisplayLead();
void CheckDisplayFormat();
void CalculateAverages();
void Spline(short *input_val, short *ret_val);
short cubic_data_arr[12][5];
short ring_data_arr[12][2];
void RingFilter(short *input_val, short *ret_val);
// short lpfilt(short datum, short init);
// Arrhythmia variables
int CalculateRWidth();
int CalculateRWidthForLead2();
int CalculateRWidthForLeadV2();
int CalculateRWidthForLeadV5();
CFont fnt;
int cnt_vpb;
int bk wnd[200], bk pntr;
int wave_width;
void CheckForWidth(double in_data, int wave_width);
void CheckForArrhythmiaType(int);
unsigned char arry byte;
int arrhythmia_type, bigem cnt;
BOOL check_for_width;
BOOL possible RonT;
short last_R_int, last_avg_RR;
short a_sys_cnt;
short arrhythmia_show_cnt;
BOOL flg 4sec;
BOOL return_val;
int qrs peak;
long sample_cnt;
short R_peak_arr[9];
short R_peak_cnt;
short turning_point, turning_point_cnt;
short hr_sum, hr_array_cnt;
short hr_array[AVG_CNT];
short R_Peak;     // value for peak of R wave
int R_amp, R_polarity;

3.10 Arrhythmia Detection

We developed an algorithm which calculates different parameters of ECG signals and an analysis is done which helps to generate different result. Using these results we can find the abnormalities in the recorded ECG. It uses Time Domain Analysis and Frequency Domain Analysis to detect the R-waves and eliminates the abnormalities in the ECG signal including the PVCs, thus the result obtained is more accurate and reliable [72]. The ECG signal is analyzed in a standardized sequence of steps to avoid missing the subtle abnormalities in the ECG tracing. After the analysis, following types of arrhythmias can be detected using the present system; Bradycardia, tachycardia, bigeminy, Trigeminy, ventricular tachycardia, RonT.

Program Implemented

void CStPlotCtrl::CheckForWidth(double in_data, int wave_width)
{
   //wave_width = CalculateRWidth();
arry_byte = arry_byte << 1;
if (wave_width >= 9)
{
    arrhythmia_type = VPC;
    return_val = VPB_WAVE;
    arry_byte |= 0x01;
    arry_byte &= 0x3f;
    switch(arry_byte)
    {
        case 0x05://101
        case 0x15://10101
        case 0x25://100101
        case 0x35://110101
        case 0x0d://1101
        case 0x2d://101101
        case 0x09://1001
        case 0x29://101001
          arrhythmia_type = BIGEMINY;
        break;
        case 0x01:
          if (possible_RonT)
              arrhythmia_type = RONT;
        break;
        case 0x0b://1011
        //case 0x1b://11011
        case 0x2b://101011
        case 0x19://11001
        case 0x33://110011
        case 0x23://100011
        case 0x13://10011
        case 0x03://11
        case 0x30://110000
        case 0x18://11000
        case 0x0c://1100
        case 0x06://110
          arrhythmia_type = COUPLET;
        break;
    }
    case 0x3a://111010
    case 0x3b://111011
case 0x07://111
case 0x17://10111
case 0x27://100111
case 0x37://110111
case 0x39://111001
case 0x2e://101110
case 0x38://111000
case 0x1c://11100
case 0x1d://11101
case 0x0e://1110
    arrhythmia_type = RUN;
break;

case 0x0f://1111
case 0x1f://11111
case 0x2f://101111
case 0x3f://111111 // Vtach
case 0x3c://111100
case 0x3d://111101
case 0x3e://111110
    arrhythmia_type = VTACH;
break;

case 0x00:
    arrhythmia_type = NORMAL;
break;

default :
    arrhythmia_type = NORMAL;
break;

} }
else
{
    arrhythmia_type = NORMAL;
return_val = R_WAVE;
arry_byte |= 0x00;
} }
int CStPlotCtrl::CalculateRWidth()
{
    int i, tmp_pntr1;
    int slp1, slp2;
    int nw1, tmp_cnt;

    tmp_pntr1 = bk_pntr - 3;
    if (tmp_pntr1 < 0) tmp_pntr1 += 100;

    nw1 = 0;
    tmp_cnt = 0;
    for (i=0; i< 30; i++)
    {
        slp1 = *(bk_wnd + tmp_pntr1 + 1) - *(bk_wnd + tmp_pntr1 + 2);
        slp2 = *(bk_wnd + tmp_pntr1 + 2) - *(bk_wnd + tmp_pntr1 + 3);
        if ((slp1 * slp2) <= 0)
        {
            if (++tmp_cnt > 1)
            {
                nw1 = i;
                break;
            }
        }
        if (--tmp_pntr1 < 0 ) tmp_pntr1 += 100;
    }
    if (!nw1) nw1 = 8;
    return nw1;
}

void CStPlotCtrl::CheckForArrhythmiaType(int wave_width)
{
    //wave_width = CalculateRWidthForLead2();
    arry_byte = arry_byte << 1;
    if (wave_width >= 9 )
    {
        arrhythmia_type = VPC;
        return_val = VPB_WAVE;
        arry_byte |= 0x01;
        arry_byte &= 0x3f;
        switch(arry_byte)
{
    case 0x05://101
        case 0x15://10101
        case 0x25://100101
        case 0x35://110101
        case 0x0d://1101
        case 0x2d://101101
        case 0x09://1001
        case 0x29://101001
            arrhythmia_type = BIGEMINY;
        break;

    case 0x01:
        if (possible_RonT)
            arrhythmia_type = RONT;
        break;

    case 0x0b://1011
        //case 0x1b://11011
        case 0x2b://101011
        case 0x19://11001
        case 0x33://110011
        case 0x23://100011
        case 0x13://10011
        case 0x03://11
        case 0x30://110000
        case 0x18://11000
        case 0x0c://1100
        case 0x06://110
            arrhythmia_type = COUPLET;
        break;

    case 0x3a://111010
    case 0x3b://111011
    case 0x07://111
    case 0x17://10111
    case 0x27://100111
    case 0x37://110111
}
case 0x39://111001
case 0x2e://101110
case 0x38://111000
case 0x1c://11100
case 0x1d://11101
case 0x0e://1110
    arrhythmia_type = RUN;
break;

case 0x0f://1111
case 0x1f://11111
case 0x2f://101111
case 0x3f://111111 // Vtach
case 0x3c://111100
case 0x3d://111101
case 0x3e://111110
    arrhythmia_type = VTACH;
break;
/*
case 0x00:
    arrhythmia_type = NORMAL;
break;

default :
    arrhythmia_type = NORMAL;
break;
*/
}
}
else
{
    arrhythmia_type = NORMAL;
return_val = R_WAVE;
arry_byte |= 0x00;
}
}

void CStPlotCtrl::DetectHR(int in_data, int Lead2_data,int LeadV2_data,int LeadV5_data)
{

short raw_start_point;
double slp1, slp2, slp3;
short Lead2_return_value = CalculateHRForLead2(in_data);
}
short CStPlotCtrl::CalculateHRForLead2(int Lead2_data)
{

double Lead2_slp2, Lead2_slp1, Lead2_slp3;
short raw_start_point;
// get the sample
rr_int_Lead2++;  
// hr calculated up to previous sample
if (Lead2_heart_rate == 0 && !(Lead2_sample_cnt % 200)) // every sec
{
    Lead2_qrs_peak *= .5;
}
if (rr_int_Lead2 > 400)
{
    arrhythmia_type = PAUSE;
    rr_int_Lead2 = 0;
    Lead2_heart_rate = 0;
    Lead2_hr_array_cnt = 0;
    Lead2_last_R_int = 0;
    number_of_peaks = 0; // increment the number of peaks
    // here manually change the lead to V5 just to check the pause
    // and rule out the possibility of very small r wave
    m_detectLead = V5;
    CheckDetectionLead();
    FireGotRWave(arrhythmia_type, Lead2_heart_rate);
    FireRRInterval(rr_int_Lead2);
}
if (++Lead2_sample_cnt >= 400)
{
    Lead2_sample_cnt = 0;
    Lead2_average_R_R = Lead2_hr_sum / AVG_CNT;
    if (Lead2_hr_sum > 0) Lead2_heart_rate = (12000 * AVG_CNT) / Lead2_hr_sum;
}
// move the data in to circular buffer
*(sld_wnd_Lead2 + 9) = Lead2_data; //ppp only lead2_data
Lead2_slp1 = *(sld_wnd_Lead2 + 9) - *(sld_wnd_Lead2 + 5);
Lead2_slp2 = *(sld_wnd_Lead2 + 5) - *(sld_wnd_Lead2 + 2);
Lead2_slp3 = *(sld_wnd_Lead2 + 5) - *(sld_wnd_Lead2 + 1);

if (Lead2_turning_point)
{
    Lead2_turning_point_cnt++;
    if(Lead2_turning_point_cnt > 10)
    {
        Lead2_turning_point_cnt = 0;
        Lead2_turning_point = 0;
    }
}

if(Lead2_arrythmia_show_cnt > -1)
{
    Lead2_arrythmia_show_cnt++;
    if(Lead2_arrythmia_show_cnt >= 400)
    {
        flg_4sec = TRUE;
        Lead2_arrythmia_show_cnt = 0;
    }
}

*(bk_wnd_Lead2 + bk_pntr_Lead2) = Lead2_data ;

if (++bk_pntr_Lead2 > 599) bk_pntr_Lead2 = 0;

if (rr_int_Lead2 > 35) //ppp 35
{
    if ((Lead2_slp1 * Lead2_slp2) < 0)   //changed from 30
    {
        if (abs(Lead2_slp2) > 20)
        {
            //if (abs(Lead2_slp3) > Lead2_qrs_peak * .25 )
            {
                //if ((int)(abs(*(sld_wnd_Lead2 + 5))) > Lead2_qrs_peak)
Lead2_qrs_peak = (int)abs(*(sld_wnd_Lead2 + 5));

// take absolute value

memcpy(Lead2_R_peak_arr,
     Lead2_R_peak_arr + 1, 8 * sizeof(short));
*(Lead2_R_peak_arr + 8) = abs(*(sld_wnd_Lead2 + 5));
++learn_count;
if(Lead2_qrs_peak > Lead2_R_Peak * 1.25 && learn_count >= 8)
{
    learn_count = 8;
    *(Lead2_R_peak_arr + 8) = Lead2_R_Peak; // add the averaged r peak
    if (peak_var == NV) // previous peak was abnormal
        peak_wave_type = NV;
    peak_var = NV;
    }
else
{
    peak_wave_type = NORMAL;
    if (peak_var == NV) // previous peak was abnormal
        peak_wave_type = NV;
    peak_var = NORMAL;
}

Lead2_R_Peak = 0;
for (int j = 0; j < 9; j++)
{
    Lead2_R_Peak += *(Lead2_R_peak_arr + j);
}
Lead2_R_Peak = Lead2_R_Peak / 9;//this is the averaged r peak

//depending on the Lead2_R_peak calculate the
    Lead2_turning_point = 0;
    Lead2_turning_point_cnt = 0;
    Lead2_check_for_width = TRUE;
if(peak_wave_type == NV
    || (rr_int_Lead2 < Lead2_average_R_R * .80 && Lead2_last_R_int != 0)
    || (rr_int_Lead2 > Lead2_average_R_R * 1.20 && Lead2_last_R_int != 0))
++err_counter;
if (rr_int_Lead2 < 45 && err_counter >= 1)
{
    // --err_counter; // checking for the noise // 27 jan 2k3
    if (err_counter > 5)
    {
        if (m_detectLead == V5)
        {
            m_detectLead = L2;
        }
        else
        {
            m_detectLead = V5;
        }
        CheckDetectionLead();
    }
    else
    {
        if (err_counter > 5) // last five beats are having same pattern
        {
            // consider r to r stable rate
            err_counter = 0;
            learn_count = 0;
            Lead2_hr_sum -= *(Lead2_hr_array + Lead2_hr_array_cnt);
            *(Lead2_hr_array + Lead2_hr_array_cnt) = rr_int_Lead2;
            Lead2_hr_sum += *(Lead2_hr_array + Lead2_hr_array_cnt);
            Lead2_hr_array_cnt++;
            if (Lead2_hr_array_cnt >= AVG_CNT) Lead2_hr_array_cnt = 0;
            Lead2_last_avg_RR = Lead2_average_R_R;
            Lead2_average_R_R = Lead2_hr_sum / AVG_CNT;
            Lead2_last_R_int = rr_int_Lead2;
            if (Lead2_hr_sum > 800 && Lead2_hr_sum < 6400)
            {
                
            }
Lead2_heart_rate = (12000 * AVG_CNT)/ Lead2_hr_sum;

prev_HR_sum = Lead2_hr_sum;

else

    
    Lead2_hr_sum -= *(Lead2_hr_array + Lead2_hr_array_cnt);
    //VPC detected do the necessary operations
    rr_int_Lead2 = Lead2_last_R_int;

    *(Lead2_hr_array + Lead2_hr_array_cnt) = rr_int_Lead2;
    Lead2_hr_sum += *(Lead2_hr_array + Lead2_hr_array_cnt);
    Lead2_hr_array_cnt++;

    if(Lead2_hr_array_cnt >= AVG_CNT) Lead2_hr_array_cnt = 0;

    if (Lead2_average_R_R < 180 )
        Lead2_last_avg_RR = Lead2_average_R_R;
    else
        Lead2_last_avg_RR = 180;

    Lead2_average_R_R = Lead2_hr_sum / AVG_CNT;

else

    
    --err_counter;
    if (err_counter < 0) err_counter = 0;
    Lead2_hr_sum -= *(Lead2_hr_array + Lead2_hr_array_cnt);
    //VPC detected do the necessary operations
    *(Lead2_hr_array + Lead2_hr_array_cnt) = rr_int_Lead2;
    Lead2_hr_sum += *(Lead2_hr_array + Lead2_hr_array_cnt);
    Lead2_hr_array_cnt++;

    if(Lead2_hr_array_cnt >= AVG_CNT) Lead2_hr_array_cnt = 0;

    if (Lead2_average_R_R < 180 )
        Lead2_last_avg_RR = Lead2_average_R_R;
    else
        Lead2_last_avg_RR = 180;
Lead2_average_R_R = Lead2_hr_sum / AVG_CNT;

Lead2_last_R_int = rr_int_Lead2;
if (Lead2_hr_sum > 800 && Lead2_hr_sum < 6400)
{
    Lead2_heart_rate = (12000 * AVG_CNT)/ Lead2_hr_sum;
    prev_HR_sum = Lead2_hr_sum;
}

Lead2_possible_RonT = FALSE;
if(Lead2_last_R_int < Lead2_last_avg_RR
}

3.11 Time Domain Analysis

The variations in heart rate may be evaluated by a number of methods. Perhaps the simplest to perform are the time domain measures. In these methods, either the heart rate at any point in time or the intervals between successive normal complexes are determined [48]. In a continuous ECG record, each QRS complex is detected, and the so-called normal-to-normal (NN) intervals (that is, all intervals between adjacent QRS complexes resulting from sinus node depolarization) or the instantaneous heart rate (IHR) is determined. Simple time domain variables that can be calculated includes the mean NN interval, the mean heart rate, the difference between the longest and shortest NN interval, the difference between night and day heart rate, and so forth. The simplest variable to calculate is the standard deviation of the NN intervals (SDNN), that is, the square root of variance. Since variance is mathematically equal to total power of spectral analysis; SDNN reflects all the cyclic components responsible for variability in the period of recording.
3.12 ECG Signal Display

Eight lead ECG signal is captured and converted to display 12 lead ECG output as per the relationships defined above. Signal is displayed on the ECG sheet defined in the form factor as per standard ECG sheet.

Program Implemented

```cpp
void CStPlotCtrl::PlotWave(short FAR* input_y)
{
    // Code using the ScreenDC & moveto lineto
    CDC *myDC;
    myDC = GetDC();
    Color cl(200,250,250,250);
    Graphics graphics(myDC->m_hDC);
    Pen t_pen(cl,2.0f);
    graphics.SetSmoothingMode(SmoothingModeAntiAlias);
    short vpb_value;
    int count = 0;

    int j;
    j = 0;
    int local_edge = dispArea.Width() - m_edge;
    // put the erase bar
    myDC->BitBlt(m_curPoint + 1, 0, 20,dispArea.Height(),&grid,
    m_curPoint + 1, 0, SRCCOPY);
    if (m_displayFormat == DISP_6_2)
    {
        myDC->BitBlt(m_curPoint + local_edge , 0,
        20,dispArea.Height(),&grid, m_curPoint + local_edge + 1, 0, SRCCOPY);
    }
    myDC->SelectObject(trace_pen);
```
if (m_displayFormat == DISP_6_2 || m_displayFormat ==
DISP_12_1)
{
    for (int i = 0; i < 6; i++)
    {
        CPoint curPoint(m_curPoint, *(m_offset + i) -
          *(input_y + i) * *(gain_arr + m_gain) / disp_scale);
        CPoint curPoint2(m_curPoint + local_edge,
          (int)(*(m_offset + i + 6) - *(input_y + i + 6) * *(gain_arr + m_gain) / disp_scale));
         /*
        myDC->MoveTo(prevPoint[i]);
        myDC->LineTo(curPoint);
        myDC->MoveTo(prevPoint[i + 6]);
        myDC->LineTo(curPoint2);
        
        // for arrythmia detection
        if(i == 1)
        {
            analysis_buffer[count] = curPoint.y;
            count ++;

            if(count = 1200)
            {
                waveDetect.ResetMe();
                vpb_value =
                waveDetect.WaveDetect(analysis_buffer,x_location,y_location);
                count = 0;
                FireShowVPB(vpb_value);
            }
        }
        */
}
graphics.DrawLine(&t_pen, prevPoint[i].x, prevPoint[i].y, curPoint.x, curPoint.y);
graphics.DrawLine(&t_pen, prevPoint[i + 6].x, prevPoint[i + 6].y, curPoint2.x, curPoint2.y);

if (flg_R)
{
    myDC->MoveTo(curPoint.x, dispArea.bottom - 5);
    myDC->LineTo(curPoint.x, dispArea.bottom - 15);
    flg_R = FALSE;
}
prevPoint[i] = curPoint;
prevPoint[i + 6] = curPoint2;

if (m_curPoint >= m_edge - 5)
{
    m_curPoint = 0;
    CRect rect;
    GetClientRect(&rect);

    for (int i = 0; i < 6; i++)
    {
        *(prevPoint + i) = 0;
        *(prevPoint + i + 6) = local_edge;
    }
    //m_prevValue = 0;
}

m_curPoint++;

else
{
}
//display format is selected three leads
for (int i = 0; i < 3; i++)
{
    switch(i)
    {

    case 0:
        j = m_displayLeads[0];
        break;

    case 1:
        j = m_displayLeads[1];
        break;

    case 2:
        j = m_displayLeads[2];
        break;
    }

    CPoint curPoint(m_curPoint, *(m_offset + i) - *(input_y + j) * *(gain_arr + m_gain) / disp_scale);
    myDC->MoveTo(prevPoint[i]);
    myDC->LineTo(curPoint);

    if (flg_R)
    {
        myDC->MoveTo(curPoint.x , dispArea.bottom - 5);
        myDC->LineTo(curPoint.x , dispArea.bottom - 15);
        flg_R = FALSE;
    }
    prevPoint[i] = curPoint;
}
if (m_curPoint >= m_edge - 10)
{

m_curPoint = 0;
CRect rect;
GetClientRect(&rect);

for (int i = 0; i < 6; i++)
{
    *(prevPoint + i) = 0;
}
m_prevValue = 0;

m_curPoint++;

if(flg_4sec)
{
    flg_4sec = FALSE;
}
ReleaseDC(myDC);

long CStPlotCtrl::GetCurPoint()
{
    return m_curPoint;
}

void CStPlotCtrl::SetCurPoint(long nNewValue)
{
    m_curPoint = nNewValue;
    SetModifiedFlag();
}

void CStPlotCtrl::showGrid(const CRect *passRect)
{
    CDC *plotDC;
    const CRect *rect = passRect;
    int i;
// now calculate the offsets
setOffset(rect);
if ((plotDC = GetDC()) == NULL) {
    AfxMessageBox("Error Initialising Screen DC");
    exit(1);
}
// assign the pointer to screen DC
plotDC->SelectStockObject(DEFAULT_PALETTE);
plotDC->RealizePalette();

if (gridBMP.m_hObject != NULL)
    gridBMP.DeleteObject();

gridBMP.CreateCompatibleBitmap(plotDC, rect->Width(), rect->Height());
if (grid.m_hDC != NULL)
    grid.DeleteDC();
grid.CreateCompatibleDC(plotDC);
CBitmap *oldBMP = grid.SelectObject(&gridBMP);
// pass the DC & rectangle for drawing the grid
CBrush *brush;
// fill the background of the display
brush = new CBrush(m_backColor);
grid.FillRect(rect, brush);
delete brush;
DrawGrid(&grid, rect);

// put the lead labels
CFont *o_Font;
oldBMP = NULL;
grid.SetTextColor(m_textColor);
o_Font = grid.SelectObject(new_font);
char *lbl[] = { " I ", " II", "III",  
"aVr", "aVI", "aVf",  
"V 1", "V 2", "V 3",  
"V 4", "V 5", "V 6"};

grid.SetBkMode(TRANSPARENT);
if (m_displayFormat == DISP_12_1 || m_displayFormat == DISP_6_2)
{
    for(i = 0; i < 6; i++)
    {
        grid.TextOut(rect->left, m_offset[i] + 5, lbl[i]);
        grid.TextOut(dispArea.Width() - m_edge, m_offset[i + 6] + 5, lbl[i + 6]);
    }
}
else
{
    for(i = 0; i < 3; i++)
    {
        grid.TextOut(rect->left, m_offset[i] + 5, lbl[m_displayLeads[i]]);
    }
}

grid.SelectObject(o_Font);
// release the plotDC so there is no memory leak
grid.SetBkMode(OPAQUE);
ReleaseDC(plotDC);

void CStPlotCtrl::setOffset(const CRect *boundRect)
{

// calculate the offsets
int spc;

// display area of the control
dispArea.top = 0;
dispArea.bottom = boundRect->Height();
dispArea.left = 0;
dispArea.right = boundRect->Width();

int i;
switch (m_displayFormat)
{
    case DISP_6_2:
        spc = boundRect->Height() / 7;
        for (i = 0; i < 6; i++)
        {
            m_offset[i + 6] = m_offset[i] = spc + spc * i;
            prevPoint[i + 6] = m_offset[i + 6];
        }
        m_edge = boundRect->Width() / 2;
        sb_page_size = (long)(350);
        break;

    case DISP_12_1:
        spc = boundRect->Height() / 13;
        for (i = 0; i < 12; i++)
        {
            m_offset[i] = spc + spc * i;
            prevPoint[i] = spc + spc * i;
        }
        m_edge = dispArea.Width();
        sb_page_size = (long)(700);
        break;
}
case DISP_3_0:
    spc = boundRect->Height() / 4;
    for (i = 0; i < 3; i++)
    {
        m_offset[i] = spc + spc * i;
        prevPoint[i] = spc + spc * i;
    }

    m_edge = dispArea.Width();
    sb_page_size = (long)(700);
    break;
}
}
void CStPlotCtrl::OnTimer(UINT nIDEvent)
{
    int ch_no = 0;
    if (nIDEvent == 1)
    {
        // put the code for splitting the data and plotting the same
        while(comm.data_cnt > 1)
        {
            if (++rd_ptr >= 999) rd_ptr = 0;
            --(comm.data_cnt);
            // send one byte to the TM if present
            if(m_i2c && TM_cnt)
            {
                CString tmp_st;
                tmp_st = TM_str[TM_str.GetLength() - TM_cnt];
                COleVariant var(tmp_st);
                comm.m_comm.SetOutput(var);
                --TM_cnt;
            }
        }
    }
}
COleControl::OnTimer(nIDEvent);
}

void CStPlotCtrl::StartTimer(long interval)
{
    SetTimer( 1,interval, NULL);
}

void CStPlotCtrl::StopTimer()
{
    KillTimer(1);
}

int CStPlotCtrl::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (COleControl::OnCreate(lpCreateStruct) == -1)
        return -1;
    // Create the comm Dialog
    comm.Create(IDD_DIALOG1);

    // assign the pointer in the CComm class of this class
    comm.StPlot = this;
    comm.EnableAutomation();

    new_font = new CFont;
    new_font->CreateFont(16,8,0,90,FW_BOLD,0,0,0,OEM_CHARSET,
        OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS ,
        DEFAULT_QUALITY ,DEFAULT_PITCH, "Arial");

    return 0;
}
short CStPlotCtrl::GetCommPort()
{
    return comm.m_comm.GetCommPort();
}

void CStPlotCtrl::SetCommPort(short nNewValue)
{
    ecg_port = nNewValue;
    try
    {
        comm.m_comm.SetCommPort(nNewValue);
    }
    catch (COleException * e)
    {
        throw (e);
    }
    SetModifiedFlag();
}

void CStPlotCtrl::StartComm()
{
    if(comm.m_comm.GetPortOpen() == FALSE)
        comm.m_comm.SetPortOpen(TRUE);
}

void CStPlotCtrl::StopComm()
{
    if(comm.m_comm.GetPortOpen() == TRUE)
        comm.m_comm.SetPortOpen(FALSE);
}
short CStPlotCtrl::GetInputLen()
{
    return comm.m_comm.GetInputLen();
}

void CStPlotCtrl::SetInputLen(short nNewValue)
{
    comm.m_comm.SetInputLen(nNewValue);
    SetModifiedFlag();
}

short CStPlotCtrl::GetRefreshInterval()
{
    return m_timer ;
}

void CStPlotCtrl::SetRefreshInterval(short nNewValue)
{
    m_timer = nNewValue;
    SetModifiedFlag();
}

void CStPlotCtrl::DoSetting()
{

    if (!m_setting)
    {
        comm.m_comm.SetInBufferCount(0);
        m_setting = TRUE;
        comm.wr_ptr = 0;
        comm.data_cnt = 0;
    }

}
void CStPlotCtrl::PutData(unsigned char *data_val)
{
short ecg_data[12], d_ecg_data[12];
short unfiltered_data[8];
short filtered_data[12];

int i,j = 0;

// get the 16 byte array from the comm port
*(unfiltered_data + 0) =
*(ecg_data + L2) = ( (long)(*(data_val + 0) * 4)
       + *(data_val + 1) >> 6 )
       - 410 ;

*(unfiltered_data + 1) =
*(ecg_data + L3) = ( (long)(*(data_val + 2) * 4)
       + *(data_val + 3) >> 6 )
       - 410 ;

j = 2;
for (i = 0; i < 6; i++)
{
    j += 2;
    *(unfiltered_data + i + 2) =
    *(ecg_data + i + 6) = ((long)(*(data_val + j) * 4)
       + *(data_val + 1 + j) >> 6 )
       - 410 ;
}
*(ecg_data + L1) = *(ecg_data + L2) - *(ecg_data + L3) ;
*(ecg_data + AVF) = (short)(*(ecg_data + L2) + *(ecg_data + L3)) / 1.732F ;
*(ecg_data + AVL) = (short)(*(ecg_data + L1) + *(ecg_data + L3)) / 1.732F ;
*(ecg_data + AVR) = (short)(-*(ecg_data + L1) + -*(ecg_data + L2)) / 1.732F;

// save the unfiltered data
if (archive != NULL)
    SaveFile(ecg_data); // save to persistence

// put the filtering routing here
Notch(unfiltered_data, filtered_data);
LowPass(filtered_data, filtered_data);

// convert to 12 leads
*(ecg_data + L2) = *(filtered_data + 0);
*(ecg_data + L3) = *(filtered_data + 1);

for (i = 0; i < 6; i++)
    *(ecg_data + i + 6) = *(filtered_data + i + 2);

// recalculate with filtered data
*(ecg_data + L1) = *(ecg_data + L2) - *(ecg_data + L3);
    *(ecg_data + AVF) = (short)(*(ecg_data + L2) + *(ecg_data + L3)) / 1.732F;
*(ecg_data + AVL) = (short)(*(ecg_data + L1) + -*(ecg_data + L3)) / 1.732F;
*(ecg_data + AVR) = (short)(-*(ecg_data + L1) + -*(ecg_data + L2)) / 1.732F;

DelayLowPass(ecg_data, d_ecg_data);
Spline(d_ecg_data, d_ecg_data);
RingFilter(d_ecg_data, d_ecg_data);

if(!auto_detect_flag)
    DetectHR(Rfilter(d_ecg_data[m_detectLead]),
               Rfilter(d_ecg_data[1]),
               Rfilter(d_ecg_data[7]),Rfilter(d_ecg_data[10]));
else
    d_ecg_data[10])),
    CLAMP(d_ecg_data[1], CLAMP_VOLTAGE, 200 ))
}
Chapter 3

Firmware Approach
Firmware Approach

1.2 Basics of Firmware

At one time, software developers who wanted to build applications to run under Windows had no choice but to write them in C. Such programs connected directly to the window application programming interface (API), a bare bone set of functions provided by the OS. In those days, besides a Windows compatible computer, the only available development tools were packaged by Microsoft into the Software Development kit, or SDK.

Learning to use the SDK to develop Windows programs has always been a difficult task. One of the reasons for this is it requires a new way of thinking. Graphical output is quite different than character based output. It takes additional time and effort to figure out that one need to anticipate all different types of input that user will send to a program. Also, one must get use to working with all the quirks in the user interface objects – windows, menus, cursors, dialogues, icons and so on – the windows provides. These build in objects let the programming be productive once a person learns how to use them. Programmers can use C++ for building windows application. With its build-in C compatibility, C++ can be used for SDK style programming [89]. However, such use does not take advantage of the object oriented features of the C++ language. Realizing the benefits of those features requires a class library that’s written to simplify the tedious housekeeping chores and frequent quirks that makes SDK programming so difficult. Several such libraries are commercially available, including Borland’s Object Windows Library (OWL) and
Microsoft’s Microsoft Foundation Class (MFC) library. The MFC library is an attempt by Microsoft to provide C++ programmers with an object oriented interface to windows. Microsoft created the MFC library with the goal of facilitating and simplifying the process of programming for Microsoft Windows.

Thus, Visual C++ is a visual development tool that makes use of the MFC library to make the development under Windows environment faster and reliable.

MFC library is a collection of C++ classes. It is provided as a Dynamic Link Library (DLL) so any application has access to the classes in MFC. A DLL consist of executable functions that are loaded into memory and are independent from any application. Libraries such as MFC are called application frameworks, because they give the user a framework for an application. The MFC classes have been built using the OS’s API functions.

The first automated ECG programs were developed in the 1970s, when digital ECG machines became possible by third generation digital signal processing boards. Commercial models, such as those developed by Hewlett Packard incorporated these programs into clinically-used devices.

During the 1980s and 1990s, extensive research was carried out by companies and by university labs in order to improve the accuracy rate, which was not very large in the first models. For this purpose, several signal databases with normal and abnormal ECGs were built up by institutions such as MIT and used to test the algorithms and its accuracy.

3.2 Phases of Holter recorder
7. A digital representation of each recorded ECG channel is obtained, by means of an analog-digital conversion device and a special data acquisition software or a digital signal processing (DSP) chip.

8. The resulting digital signal is processed by a series of specialized algorithms, which start by conditioning it, e.g., removal of noise, baseline variation, etc.

9. **Feature extraction:** mathematical analysis is now performed on the clean signal of all channels, to identify and measure a number of features which are important for interpretation and diagnosis, this will constitute the input to AI-based programs, such as the peak amplitude, area under the curve, displacement in relation to baseline, etc, of the P, Q, R, S and T waves, the time delay between these peaks and valleys, heart rate frequency (instantaneous and average), and many others. Some sort of secondary processing such as Fourier analysis and wavelet analysis may also be performed in order to provide input to pattern recognition-based programs.

10. Logical processing and pattern recognition, using rule-based expert systems, probabilistic Bayesian analysis or fuzzy logics algorithms, cluster analysis, artificial neural networks, genetic algorithms and others techniques are used to derive conclusions, interpretation and diagnosis

11. A reporting program is activated and produces a proper display of original and calculated data, as well as the results of automated interpretation.

12. In some applications, such as automatic defibrillators, an action of some sort may be triggered by results of the analysis, such as the occurrence of an atrial fibrillation or a cardiac arrest, the sounding of alarms in a medical monitor in intensive-care unit applications, and so on.
3.11 Analyzing Software of Holter Recorder

When the recording of ECG signal is finished (usually after 24 or 48 hours), it is up to the physician to perform the signal analysis. Since it would be extremely time demanding to browse through such a long signal, there is an integrated automatic analysis process in each Holter software which automatically determines different sorts of heart beats, rhythms, etc. However the success of the automatic analysis is very closely associated with the signal quality. The quality itself mainly depends on the attachment of the electrodes to the patient body. If these are not properly attached, the electromagnetic disturbance surrounding us will influence the ECG signal resulting thus in a very noisy record. If the patient moves rapidly, the distortion will be even bigger. Such record is then very difficult to process. Besides the attachment and quality of electrodes, there are another factors affecting the signal quality, such as muscle tremors, sampling rate and resolution of the digitalized signal (high quality devices offer 2000Hz and 16 bits or higher) [28].

The automatic analysis commonly provides the physician with information about heart beat morphology, beat interval measurement, heart rate variability, rhythm overview and patient diary (moments when the patient pressed the patient button). Advanced systems also perform the spectral analysis, ischemic burden evaluation, graph of patient’s activity or PQ segment analysis. Another requirement is the ability of pacemaker detection and analysis. Such ability is useful when one wants to check the correct pacemaker function.
3.12 ECG Analysis System Software

3.12.1 Interpretation of ECG

Interpretation of 12 lead ECG using software or computers uses algorithms to determine whether a patient is normal or abnormal. It helps in determining the abnormalities if discovered. Right back from era of 1960s’ when some mainframe computers centrally located in computing centers performed the ECG analysis to the modern period ECG analysis machines that are minicomputer based; interpretation of the ECG signal has improved to a great extent. Today’s machines help perform the complete data acquisition, processing and interpretation of the ECG signal at the patient’s bedside without transmitting any data to remote computer [90].

The modern microprocessor based ECG interpretive machines include eight ECG amplifiers so that they can simultaneously sample and store eight leads. They then synthesize the other four redundant leads. These machines include enough memory to store all leads for a 10 second interval at a clinical sampling rate of 250sps – 500sps. ECG interpretation starts with feature extraction, which has two parts; 1) Waveform recognition to identify the waves in the ECG signal and, 2) measurement to quantify a set of amplitudes and time durations that is to be used to drive the decision making process. The decision is made based in comparison to the standard values that are fed in the computer on which it operates.
We propose a system that performs all the basic tasks involved in interpretation of the 12 lead ECG signal and at the same time performs various other functions which involves data management from previous cases, maintaining log, transmission of information, etc.

The analysis system defined performs complete 12-lead ECG interpretation. The computer approaches the ECG interpretation as a pattern classification problem and applies a multivariate statistical pattern recognition method to solve it. The system mimics the human expert’s decision process using a rule based expert system. During the ECG analysis the system performs various mathematical tasks to extract and calculate the different features of the ECG signal. It calculates the P wave and T wave interval, detects QRS complex and measures its interval. The system determines the RR peak interval. It looks out for any missing or abnormal waveforms. Finally it predicates the most possible arrhythmia from the derived parameters using the predefined condition. In our development, we gave importance in designing a system to analyze ECG waveform. The analyzing of the signal helps in understanding various patterns of ECG signal and execute appropriate decision about the health of the patient. This system deals with implementation of algorithms like detrending of ECG signal, RR interval detection, QRS complex detection, detection of pre ventricular contractions and other abnormalities using embedded controllers. The main component of the firmware includes;

3. Software using High level language C++ to analyze it
4. GUI designed using Visual basic
As the main aim was to develop an embedded system, algorithms for determination of RR interval, detrending of RR waveform, QRS detection, and Beat Rate detection were designed and implemented in higher level language. The higher level languages used in the applications are C++ and Visual Basics. C++ is used for calculative and numerical reasoning including decision making whereas Visual basics is used for designing HMI and displaying the main result and waveform on the console.
3.13 System Flow Algorithm

ECG analysis system is designed to function as follows:

Whenever it is started it asks whether the patient is new or old.

NOW,

9 If the patient is new, then the system gives privilege to fill details of the new patient and register it. This is done so that his/her ECG waveform will be directly stored in his/her record which can be accessed for future reference. Every patient is given an own unique numerical ID.

10 If the patient was previously registered then we can select the register option to check for his records present in the system.

11 Once the patient is registered, the system checks the serial ports to capture the ECG waveform received by the hardware and display it on the screen.

12 Before displaying the data, the system performs some set of operations in order to remove noise, smoothens the signal, etc.

13 Once the data is displayed, simultaneously, it analyzes the acquired waveform to display heart rate.

14 It calculates heart rate, R wave width or signal width, detects QRS complex.
Using the analyzed data, ECG analysis system checks through the algorithm for a probable arrhythmia. This is important and unique feature available in the system to display the type of arrhythmia which will helps the doctor to take the fast remedial action and the derived arrhythmia is displayed on the screen.

The output of ECG analysis system i.e. the status of the patient Normal/abnormal along with types of arrhythmia can then be messaged using GSM module. This is a unique feature available in the analysis system. Transmission of ECG analysis output is very important.
Figure 3.1: System Flow Algorithm
3.14 Computing [Signal Processing]

Computing is usually defined as the activity of using and improving computer technology, computer hardware and software. It is the computer-specific part of information technology. Computer science (or computing science) is the study and the science of the theoretical foundations of information and computation and their implementation and application in computer systems.

Computing Curricula 2005 defined computing: [37]

In a general way, we can define computing to mean any goal-oriented activity requiring, benefiting from, or creating computers. Thus, computing includes designing and building hardware and software systems for a wide range of purposes; processing, structuring, and managing various kinds of information; doing scientific studies using computers; making computer systems behave intelligently; creating and using communications and entertainment media; finding and gathering information relevant to any particular purpose, and so on. The list is virtually endless, and the possibilities are vast.

A computer is a machine that manipulates data according to a set of instructions called a computer program. The program has an executable form that the computer can use directly to execute the instructions. The same program in its human-readable source code form, enables a programmer to study and develop the algorithm. Because the instructions can be carried out in different types of computers, a single set of source
instructions converts to machine instructions according to the central processing unit type.

The execution process carries out the instructions in a computer program. Instructions express the computations performed by the computer. They trigger sequences of simple actions on the executing machine. Those actions produce effects according to the semantics of the instructions.

Computer programming in general is the process of writing, testing, debugging, and maintaining the source code and documentation of computer programs. This source code is written in a programming language, which is an artificial language, restrictive, demanding, and unforgiving to humans but easily translated by the computer. The purpose of programming is to invoke the desired behavior (customization) from the machine. The process of writing high quality source code requires knowledge of both the application's domain and the computer science domain. The highest quality software is thus developed by a team of various domain experts, each person a specialist in some area of development. But the term programmer may apply to a range of program quality, from hacker to open source contributor to professional [89]. And a single programmer could do most or all of the computer programming needed to generate the proof of concept to launch a new "killer" application.
3.6.1 Programming languages

Different programming languages support different styles of programming (called *programming paradigms*). The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference. Ideally, the programming language best suited for the task at hand will be selected. Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of compilers for that language, and the efficiency with which programs written in a given language execute.

The details look different in different languages, but a few basic instructions appear in just about every language:

- **input**: Get data from the keyboard, a file, or some other device.
- **output**: Display data on the screen or send data to a file or other device.
- **arithmetic**: Perform basic arithmetical operations like addition and multiplication.
- **conditional execution**: Check for certain conditions and execute the appropriate sequence of statements.
- **repetition**: Perform some action repeatedly, usually with some variation.

Many computer languages provide a mechanism to call functions provided by libraries. Provided the functions in a library follow the appropriate run time conventions (e.g., method of passing arguments), then these functions may be written in any other language.
3.6.2 Modern programming

Whatever the approach to software development may be, the final program must satisfy some fundamental properties. The following properties are among the most relevant:

- **Efficiency/performance**: the amount of system resources a program consumes (processor time, memory space, slow devices such as disks, network bandwidth and to some extent even user interaction): the less, the better. This also includes correct disposal of some resources, such as cleaning up temporary files and lack of memory leaks.

- **Reliability**: how often the results of a program are correct. This depends on conceptual correctness of algorithms, and minimization of programming mistakes, such as mistakes in resource management (e.g., buffer overflows and race conditions) and logic errors (such as division by zero).

- **Robustness**: how well a program anticipates problems not due to programmer error. This includes situations such as incorrect, inappropriate or corrupt data, unavailability of needed resources such as memory, operating system services and network connections, and user error.

- **Usability**: the ergonomics of a program: the ease with which a person can use the program for its intended purpose, or in some cases even unanticipated purposes. Such issues can make or break its success even regardless of other issues. This involves a wide range of textual, graphical and sometimes hardware elements that improve the clarity, intuitiveness, cohesiveness and completeness of a program's user interface.

- **Portability**: the range of computer hardware and operating system platforms on which the source code of a program can be
compiled/interpreted and run. This depends on differences in the programming facilities provided by the different platforms, including hardware and operating system resources, expected behavior of the hardware and operating system, and availability of platform specific compilers (and sometimes libraries) for the language of the source code.

- **Maintainability**: the ease with which a program can be modified by its present or future developers in order to make improvements or customizations, fix bugs and security holes, or adapt it to new environments. Good practices during initial development make the difference in this regard. This quality may not be directly apparent to the end user but it can significantly affect the fate of a program over the long term.

### 3.6.3 Algorithmic complexity

The academic field and the engineering practice of computer programming are both largely concerned with discovering and implementing the most efficient algorithms for a given class of problem. For this purpose, algorithms are classified into orders using so-called Big O notation, $O(n)$, which expresses resource use, such as execution time or memory consumption, in terms of the size of an input. Expert programmers are familiar with a variety of well-established algorithms and their respective complexities and use this knowledge to choose algorithms that are best suited to the circumstances.
3.15 Software Implemented

3.15.1 Determination of RR interval

ECG was recorded continuously, during a passive event related potential paradigm, where subject sat in a chair while auditory pitch stimuli were delivered to right ear. Discrete event series, \( R_i = R_i - 1 \) intervals as a function of \( R_i \) occurrence times, was constructed by an adaptive QRS detector algorithm. As a result of the detection algorithm an unevenly sampled RR interval time series was obtained. In order to recover an evenly sampled signal from the irregularly sampled event series cubic interpolation was applied. Sampling rate of the ECG was 350 Hz. Discrete event series, \( R_i - R_{i-1} \) intervals as a function of \( R_i \) occurrence times, was constructed by an adaptive QRS detector algorithm. As a result of the detection algorithm an unevenly sampled RR interval time series was obtained. In order to recover an evenly sampled signal from the irregularly sampled event series cubic interpolation was applied. [41]

We denote the RR interval time series as

\[
Z = (R2 - R1; R3 - R2 ; \ldots \ldots RN - R_{N-1})^T \in \mathbb{R}^{N-1}
\]

where \( N \) is the number of R peaks detected. The RR series can be considered to consist of two components

\[
Z = Z_{\text{stat}} + Z_{\text{trend}}
\]
where \( z_{\text{stat}} \) is the nearly stationary RR series of interest and \( z_{\text{trend}} \) is the low frequency aperiodic trend component. The trend component can be modeled with a linear observation model as

\[
z_{\text{trend}} = H\theta + \nu
\]

where \( H \in \mathbb{R}^{(N-1) \times M} \) is the observation matrix, \( \theta \in \mathbb{R}^M \) are the regression parameters and \( \nu \) is the observation error. The task is then to estimate the parameters by some fitting procedure so that the prediction \( \hat{z}_{\text{trend}} = H\hat{\theta} \) be used as the estimate of the trend. The properties of the estimate depend strongly on the properties of the basis vectors (columns of the matrix \( H \)) in the fitting. Widely used method for the solution of the estimate \( \hat{\theta} \) is the least squares method.

**Program implemented:**

```cpp
// DetectR.cpp: implementation of the DetectR class.
#include "stdafx.h"
#include "StPlot.h"
#include "DetectR.h"
#include "math.h"
#ifdef _DEBUG
#undef THIS_FILE
#define THIS_FILE=__FILE__;
#endif

float max_array(float wkept[], int len);
float min_array(float min_1, float min_2);
```
int value = 0, step = 0;
int R_count = 0;
int cn_t;
int r_off;
static xy;
int outputInt = 0;
#define scale 3
#define xmax 1.0039
#define dx 0.0039
#define prod_ScaleDx (int)(scale * dx)
#define prod_ScaleXMax (int)(scale * xmax)
#define Rate 100
#define tra 4
#define data_length 400 // (Rate * 4)
#define Resultant_length (int)(400 + prod_ScaleXMax)
float K[prod_ScaleXMax];
float fun_1[Resultant_length];
float Diff[Resultant_length];
float Conv[Resultant_length];
float wkept[Resultant_length];
int timecounter = 0, vpb_per_min = 0;
int abnormal_count = 0, normal_count = 0, set_detect_flag = 0;
int vpb_pointer = 0;

DetectR::DetectR()
{
}
short DetectR::WaveRDetect(double * inputSignal, double * outputX_Array, double * outputY_Array)
{
    R_count = 0;
    cn_t = 0;
    for(int i = 0; i < 1200; i++)
    {
        inputInt[i] = (int)* (inputSignal + i);
    }
    DetectRWave(inputInt, outputIntX, outputIntY);
    for(i = 0; i <= R_count; i++)
    {
        *(outputX_Array + i) = (double) outputIntX[i];
        *(outputY_Array + i) = (double) outputIntY[i];
    }

    return R_count;
}

void DetectR::ResetMe()
{
    for(int i = 0; i < prod_ScaleXMax; i++)
    {
    }
for(i = 0; i < Resultant_length; i++) {
    Conv[i] = 0;
}

for(i = 0; i < 1200; i++) {
    outputIntX[i] = 0;
    outputIntY[i] = 0;
}

R_count = 0;
if (timecounter == 15) {
    timecounter = 0;
    vpb_per_min = 0;
}

short DetectR::WaveDetect(short * input, short * outputX, short * outputY) {
    R_count = 0;
    cn_t = 0;
int Q_value = 0, S_value = 0, VPB_value = 0, k = 0;
int sampling = 3, op_width = 0;
for(int i = 0; i < 1200; i++)
{
    if(sampling == 3)
    {
        sampling = 0;
        inputInt[k] = (int)(*(input + i) / 5);
        k++;
    }
    sampling++;
}
DetectRWave(inputInt, outputIntX, outputIntY);
for(i = 0; i <= abs(R_count); i++)
{
    *(outputX + i) = (int)outputIntX[i];
    *(outputY + i) = (int)outputIntY[i];
    op_width = FindQRS((int)outputIntX[i], Q_value, S_value);
    *(outputY + i) = op_width;
}
timecounter ++;
return (R_count + 1);
}
void DetectR::DetectRWave(int *signal, int *x_location, int *y_location)
{
    int i = 0, j = 0, k = 0, first = 0, scnd = 0, count = 0;
    unsigned char check = 0;
int x_pos[100], x_lock[30];
float least_value = -1000.0;
int timer, final = 0, high_value = 2000, low_value = -2000, final_value = -3200, qrs_final = 0;
int greater_value = 1000;
float multi, optim_ze = 0;

fun_1[0] = 0;
fun_1[1] = 0.3360;
fun_1[2] = 0.3320;
fun_1[3] = 0;

for (i = 0; i < (Resultant_length); i++)
{
    if (i >= data_length)
        signal[i] = 0;

    if (i >= prod_ScaleXMax)
        fun_1[i] = 0;
}

multi = (float) sqrt(scale);
for (k = 0; k < 400; k++)
{
    if (k <= 1)
    {
    
}
optim_ze = (*(signal + j)) * (*(fun_1 + k - j));
Conv[k] = Conv[k] + optim_ze;

else
{
    for( j = k - 2; j < k; j++)
    {
        optim_ze = (*(signal + j)) * (*(fun_1 + k - j));
        Conv[k] = Conv[k] + optim_ze;
    }
}

if(k >0 )
{
    Conv[k - 1] = -(Conv[k - 1])-(Conv[k]); // * multi;
    if(Conv[k - 1] != 0)
    {
        Conv[k - 1] = Conv[k - 1] * multi;
        if (abs(Conv[k - 1]) > least_value)
            least_value = abs(Conv[k - 1]);
    }
}

j = 0;
for (first = 10;first < data_length;first++)
{
    if (abs(Conv[first]) > least_value * 0.35)
{ 
    x_pos[count] = first;
    count++;
}

// sorting the real R points from the above points.
for (scnd = 0; scnd <= count - 1; scnd++)
{
    if (x_pos[scnd + 1] - x_pos[scnd] > 20)
    {
        x_lock[check] = x_pos[scnd];
        x_lock[check + 1] = x_pos[scnd + 1];
        check++;
    }
}
final = 0;
if (check == 0)
    return;
for (scnd = 0; scnd <= check; scnd++)
{
    for (timer = x_lock[scnd] - (int)(Rate * 0.2); timer < x_lock[scnd] + (int)(Rate * 0.03); timer++)
    {
        if (abs(signal[timer]) >= final_value)
            goto label;
        if (abs(signal[timer]) >= final_value)
x_location[final] = timer;
y_location[final] = signal[timer];
final_value = abs(signal[timer]);

label:;

final++;
final_value = -3200;

if (final == 0)
R_count = final;
else
R_count = final - 1;

int DetectR::FindBeatStart(int Rpoint)
{
int Mult;
int i,slope_1,slope_2,amplitude,FindStartOfWave;

if(Rpoint < (int)(Rate * 0.03))
Rpoint = (int)(Rate * 0.03);
amplitude = abs(inputInt[Rpoint]);
for (i = Rpoint - (int)(Rate * 0.03); i > Rpoint - (int)(Rate * 0.1);i--)
{
  slope_1 = inputInt[i + 2] - inputInt[i + 1];
slope_2 = inputInt[i + 1] - inputInt[i];
Mult = slope_1 * slope_2;
if (Mult <= 0)
    { 
        FindStartOfWave = i + 1;
        break;
    }

FindStartOfWave = i + 1;
return(FindStartOfWave);
}

int DetectR::FindBeatEnd(int Rpoint)
{
    int i,FindEndOfWave,slope_1,slope_2;
    int Mult;
    int amplitude;
    amplitude = abs(inputInt[Rpoint]);
    for (i = Rpoint ; i < Rpoint + (int)(Rate * 0.10);i++)
        {
            if (i >= Rate * tra)
                break;
            else
                {
                    slope_1 = inputInt[i + 2] - inputInt[i + 1];
                    slope_2 = inputInt[i + 1] - inputInt[i];
                    Mult = slope_1 * slope_2;
                    if (Mult <= 0)
FindEndOfWave = i + 1;
break;
}

FindEndOfWave = i + 1;
return(FindEndOfWave);

int DetectR::FindQRS(int R_value, int Q_x, int S_x)
{
    int width = 0;
    Q_x = FindBeatStart(R_value);
    S_x = FindBeatEnd(R_value);
    width = S_x - Q_x;
    VPB_Analysis(width);
    return(width);
}

void DetectR::VPB_Analysis(int QS_width)
{
    if (QS_width > Rate * 0.1 && QS_width < Rate * 3)
    {
        if (abnormal_count > 0 && normal_count > 0)
            set_detect_flag = 1;
        else
            abnormal_count = abnormal_count + 1;
    }
if (abnormal_count > 14)
    set_detect_flag = 1;

vpb_per_min = vpb_per_min + 1;
}

else
{
    if (abnormal_count > 0)
        normal_count = normal_count + 1;

    if (abnormal_count > 1 && normal_count > 1)
        set_detect_flag = 1;
}

if (set_detect_flag == 1)
{
    if (abnormal_count == 1 && normal_count == 1)
        vpb_pointer = 1; //"Bigeminy"
    else if (abnormal_count == 1 && normal_count == 2)
        vpb_pointer = 2; //"Trigeminy"
    else if (abnormal_count == 2 && normal_count > 1)
        vpb_pointer = 3; //"Pair PVC"
    else if (abnormal_count == 5 && normal_count > 1)
        vpb_pointer = 4; //"Run 5 PVC"
    else if (abnormal_count == 11 && normal_count > 1)
        vpb_pointer = 5; //"Run 11 PVC"
else if (abnormal_count > 14)
    vpb_pointer = 6;  //" V.Tach"
    abnormal_count = 0;
normal_count = 0;
set_detect_flag = 0;
}
if (timecounter == 14)
{
    if (vpb_per_min == 24)
        vpb_pointer = 7;
    else if  (vpb_per_min == 12)
        vpb_pointer = 8;
    else if  (vpb_per_min == 6)
        vpb_pointer = 9;
}

3.7.2 PSD Estimation

Power spectral density (PSD) analysis provides the basic information of how power (variance) distributes as a function of frequency. Independent of the method used, only an estimate of the true PSD of the signal can be obtained from proper mathematical algorithms. [48]

Methods for PSD estimation can be classified as nonparametric (e.g. methods based on FFT) and parametric (methods based on autoregressive
(AR) time series modelling). In the latter approach the RR time series is modelled as an AR (p) process.

\[ Z_t = - \sum a_j Z_{t-j} + e_t, \quad t = p+1, \ldots, N-1 \quad \& \quad 1 \leq j \geq p \]

where \( p \) is the model order, \( a_j \) are the AR coefficients and \( e_t \) is the noise term. A modified covariance method is used to solve the AR model. The power spectrum estimate \( P_z \) is then calculated as:

\[ P_z(\omega) = \sigma^2 / \left| 1 + \sum a_j e^{-i\omega j} \right|^2 \]

### 3.7.3 Welch’s Method

Welch's method (also called the periodogram method) for estimating power spectra is carried out by dividing the time signal into successive blocks, forming the periodogram for each block, and averaging. Denote the \( m \)th windowed, zero-padded frame from the signal \( x \) by

\[ x_m(n) \triangleq w(n)x(n + mR), \quad n = 0, 1 \ldots M-1; \quad m = 0, 1 \ldots K-1 \]

Where \( R \) is defined as the window hop size, and let \( K \) denote the number of available frames. The Welch estimate of the power spectral density is given by

\[ \hat{S}_x^W(\omega_k) \triangleq \frac{1}{K} \sum_{m=0}^{K-1} P_{x_m, M}(\omega_k). \]

In frequency-domain the PSD is analyzed by calculating powers and peak frequencies for different frequency bands. The commonly used frequency
bands are very low frequency (VLF, 0-0.04 Hz), low frequency (LF, 0.04-0.15 Hz), and high frequency (HF, 0.15-0.4 Hz).

3.7.4 QRS Detection

The precision in the identification of QRS complexes is of great importance for the reliability of an automated ECG analyzing system and thus, for the diagnosis of cardiac diseases. [24]

The proposed method can be divided into three steps: 1) estimation of the initial R-wave fiducial points, 2) extraction of R-wave data points and modeling of the R-wave shape, and 3) correction of the fiducial point values using the estimated model. The second and last steps can be repeated until the fiducial point values converge. An illustrative diagram of the proposed method is presented in figure above and the three steps are described in details in the following.

Stepwise representation of QRS detection on algorithm;
Figure 3.2 QRS Detection Algorithm

**Step 1: Estimation of initial R-Wave Fiducial points**

At First, an adaptive QRS detector algorithm based on the one presented in is applied to detect the R-wave fiducial points from the sparsely sampled ECG recording. The accuracy of the observed time instants is then improved by QRS interpolation (a piecewise cubic spline interpolation with a sampling rate equal to 20 kHz). The observed R-wave maximums after interpolation are taken as initial guesses for R-wave fiducial points.

**Step 2: Modeling the R-wave shape**
Once the initial R-wave fiducial points have been estimated the shape of R-wave is modeled. For this, data points from each R-wave are extracted by using a time window centered at the corresponding initial fiducial point values. The data points of each R-wave are then accumulated into analogous temporal frame. The analogous temporal frame is obtained by subtracting the initial fiducial point instants from the time indices of the corresponding R-wave data points.

**Step 3: Correction of the R-wave fiducial point values**

The final step of the proposed method is to use the estimated R-wave model for correcting the initial fiducial point time instants. This is accomplished through linear LS regression. As regressors we select the estimated model $h(\hat{\theta}, t)$ and its first derivative $dh(\hat{\theta}; t)/dt$. The derivative is included in the regression to enable the shifting of the peak position in time.

### 3.16 Heart Rate Calculation

**Heart rate** is the number of heartbeats per unit of time - typically expressed as beats per minute (bpm) - which can vary as the body’s need for oxygen changes, such as during exercise or sleep [79]. The measurement of heart rate is used by medical professionals to assist in the diagnosis and tracking of medical conditions. It is also used by individuals, such as athletes, who are interested in monitoring their heart rate to gain
maximum efficiency from their training. The **R wave to R wave interval** (RR interval) is the inverse of the heart rate.

Heart rate is measured by finding the pulse of the body. This pulse rate can be measured at any point on the body where an artery's pulsation is transmitted to the surface - often as it is compressed against an underlying structure like bone - by pressuring it with the index and middle finger. The thumb should not be used for measuring another person's heart rate, as its strong pulse may interfere with discriminating the site of pulsation. [80]

### 3.17 Detection of Abnormal Waveforms

Detection of abnormal waveform and missing waveform in the ECG signal is of great importance. Abnormal waveforms such as Pre-Ventricular Contraction (PVC), if present, can lead to error in the analysis. To detect these abnormal waveforms, we detect the QRS complexes in the ECG signal and then decompose it into set of frequency bands using wavelet transformation. To distinguish between “true” R-waves and PVCs, an adaptive threshold is implemented with a value greater than that of R-waves and less than the value of PVCs. After identifying the PVCs, they are eliminated with other aberrations in the signal in order to produce an R-wave.

**Program Implementation**

// Abnormal Waveform
short points_to_add, average_offset, average_initialised, number_of_peaks;
short possible_R, clamp_cnt, max_data_point, R_peak_found;
short sec_6_cnt, average_R_R;
void DetectHR(double, short); // detect VPB
void DetectHR(int, int, int, int);  // detect only HR
//void DetectHR(int);    // detect only HR
//void DetectHR(int,int,int,int);
short CalculateHRForLead2(int);
short CalculateHRForLeadV2(int);
short CalculateHRForLeadV5(int);

void CheckDetectionLead();
void CheckDisplayLead();
void CheckDisplayFormat();
void CalculateAverages();
void Spline(short *input_val, short *ret_val);
short cubic_data_arr[12][5];
short ring_data_arr[12][2];
void RingFilter(short *input_val, short *ret_val);
// short lpfilt(short datum, short init);
// Arrhythmia variables
int CalculateRWidth();
int CalculateRWidthForLead2();
int CalculateRWidthForLeadV2();
int CalculateRWidthForLeadV5();
CFont fnt;
int cnt_vpb;
int bk_wnd[200], bk_pntr;
int wave_width;
void CheckForWidth(double in_data, int wave_width);
void CheckForArrhythmiaType(int);
unsigned char arry_byte;
int arrhythmia_type, bigem_cnt;
BOOL check_for_width;
BOOL possible_RonT;
short last_R_int, last_avg_RR;
short a_sys_cnt;
short arrythmia_show_cnt;
BOOL flg_4sec;
BOOL return_val;
int qrs_peak;
long sample_cnt;
3.18 Arrhythmia Detection

We developed an algorithm which calculates different parameters of ECG signals and an analysis is done which helps to generate different result. Using these results we can find the abnormalities in the recorded ECG. It uses Time Domain Analysis and Frequency Domain Analysis to detect the R-waves and eliminates the abnormalities in the ECG signal including the PVCs, thus the result obtained is more accurate and reliable [72]. The ECG signal is analyzed in a standardized sequence of steps to avoid missing the subtle abnormalities in the ECG tracing. After the analysis, following types of arrhythmias can be detected using the present system; Bradycardia, tachycardia, bigeminy, Trigeminy, ventricular tachycardia, RonT.

Program Implemented

```cpp
void CStPlotCtrl::CheckForWidth(double in_data, int wave_width) {
    //wave_width = CalculateRWidth();
```
arry_byte = arry_byte << 1;
if (wave_width >= 9 )
{
    arrhythmia_type = VPC;
    return_val = VPB_WAVE;
    arry_byte |= 0x01;
    arry_byte &= 0x3f;
    switch(arry_byte)
    {
      case 0x05://101
      case 0x15://10101
      case 0x25://100101
      case 0x35://110101
      case 0x0d://1101
      case 0x2d://101101
      case 0x09://1001
      case 0x29://101001
      arrhythmia_type = BIGEMINY;
      break;
      case 0x01:
      if (possible_RonT)
          arrhythmia_type = RONT;
      break;
      case 0x0b://1011
        //case 0x1b://11011
      case 0x2b://101011
      case 0x19://11001
      case 0x33://110011
      case 0x23://100011
      case 0x13://10011
      case 0x03://11
      case 0x30://110000
      case 0x18://11000
      case 0x0c://1100
      case 0x06://110
      arrhythmia_type = COUPLE;
      break;
      case 0x3a://111010
      case 0x3b://111011

case 0x07://111
case 0x17://10111
case 0x27://100111
case 0x37://110111
case 0x39://111001
case 0x2e://101110
case 0x38://111000
case 0x1c://11100
case 0x1d://11101
case 0x0e://1110
    arrhythmia_type = RUN;
break;

case 0x0f://1111
case 0x1f://11111
case 0x2f://101111
case 0x3f://111111
    // Vtach
case 0x3c://111100
case 0x3d://111101
case 0x3e://111110
    arrhythmia_type = VTACH;
break;

case 0x00:
    arrhythmia_type = NORMAL;
break;

default :
    arrhythmia_type = NORMAL;
break;

} }
else 
{
    arrhythmia_type = NORMAL;
return_val = R_WAVE;
arr_byte |= 0x00;
} }
int CStPlotCtrl::CalculateRWidth()
{
    int i, tmp_pntr1;
    int slp1, slp2;
    int nw1, tmp_cnt;

    tmp_pntr1 = bk_pntr - 3;
    if (tmp_pntr1 < 0) tmp_pntr1 += 100;

    nw1 = 0;
    tmp_cnt = 0;
    for (i=0; i< 30; i++)
    {
        slp1 = *(bk_wnd + tmp_pntr1 + 1) - *(bk_wnd + tmp_pntr1 + 2);
        slp2 = *(bk_wnd + tmp_pntr1 + 2) - *(bk_wnd + tmp_pntr1 + 3);
        if ((slp1 * slp2) <= 0)
        {
            if (++tmp_cnt > 1)
            {
                nw1 = i;
                break;
            }
        }
        if (--tmp_pntr1 < 0 ) tmp_pntr1 += 100;
    }
    if (!nw1) nw1 = 8;
    return nw1;
}

void CStPlotCtrl::CheckForArrhythmiaType(int wave_width)
{
    //wave_width = CalculateRWidthForLead2();
    arry_byte = arry_byte << 1;
    if (wave_width >= 9 )
    {
        arrhythmia_type = VPC;
        return_val = VPB_WAVE;
        arry_byte |= 0x01;
        arry_byte &= 0x3f;
        switch(arry_byte)
{  
case 0x05://101  
case 0x15://10101  
case 0x25://100101  
case 0x35://110101  
case 0x0d://1101  
case 0x2d://101101  
case 0x09://1001  
case 0x29://101001  
    arrhythmia_type = BIGEMINY;  
break;  

case 0x01:  
    if (possible_RonT)  
        arrhythmia_type = RONT;  
break;  

case 0x0b://1011  
//case 0x1b://11011  
case 0x2b://101011  
case 0x19://11001  
case 0x33://110011  
case 0x23://100011  
case 0x13://10011  
case 0x03://11  
case 0x30://110000  
case 0x18://11000  
case 0x0c://1100  
case 0x06://110  
    arrhythmia_type = COUPLET;  
break;  

case 0x3a://111010  
case 0x3b://111011  
case 0x07://111  
case 0x17://10111  
case 0x27://100111  
case 0x37://110111}
case 0x39://111001
    arrhythmia_type = RUN;
    break;

case 0x2e://101110

case 0x38://111000

case 0x1c://11100

case 0x1d://11101

case 0x0e://1110
    arrhythmia_type = RUN;
    break;

case 0x0f://1111

case 0x1f://11111

case 0x2f://101111
    case 0x3f://111111 // Vtach

case 0x3c://111100

case 0x3d://111101

case 0x3e://111110
    arrhythmia_type = VTACH;
    break;
    /*
    case 0x00:
        arrhythmia_type = NORMAL;
        break;
    
    default :
        arrhythmia_type = NORMAL;
        break;
    */
    
}
}
else
{
    arrhythmia_type = NORMAL;
    return_val = R_WAVE;
    arry_byte |= 0x00;
}
}

void CStPlotCtrl::DetectHR(int in_data, int Lead2_data,int LeadV2_data,int LeadV5_data)
{

short raw_start_point;
double slp1, slp2, slp3;
short Lead2_return_value = CalculateHRForLead2(in_data);
}
short CStPlotCtrl::CalculateHRForLead2(int Lead2_data)
{
    double Lead2_slp2, Lead2_slp1, Lead2_slp3;
    short raw_start_point;
    // get the sample
    rr_int_Lead2++;
    // hr calculated up to previous sample
    if (Lead2_heart_rate == 0 && !(Lead2_sample_cnt % 200)) // every sec
    {
        Lead2_qrs_peak *= .5;
    }
    if (rr_int_Lead2 > 400)
    {
        arrhythmia_type = PAUSE;
        rr_int_Lead2 = 0;
        Lead2_heart_rate = 0;
        Lead2_hr_array_cnt = 0;
        Lead2_last_R_int = 0;
        number_of_peaks = 0; // increment the number of peaks
        // here manually change the lead to V5 just to check the pause
        // and rule out the possibility of very small r wave
        m_detectLead = V5;
        CheckDetectionLead();
        FireGotRWave(arrhythmia_type, Lead2_heart_rate);
        FireRRInterval(rr_int_Lead2);
    }
    if (++Lead2_sample_cnt >= 400)
    {
        Lead2_sample_cnt = 0;
        Lead2_average_R_R = Lead2_hr_sum / AVG_CNT;
        if (Lead2_hr_sum > 0) Lead2_heart_rate = (12000 * AVG_CNT) / Lead2_hr_sum;
    }
    // move the data in to circular buffer
*(sldWnd_Lead2 + 9) = Lead2_data;  // pppp only lead2_data

Lead2_slp1 = *(sldWnd_Lead2 + 9) - *(sldWnd_Lead2 + 5);
Lead2_slp2 = *(sldWnd_Lead2 + 5) - *(sldWnd_Lead2 + 2);
Lead2_slp3 = *(sldWnd_Lead2 + 5) - *(sldWnd_Lead2 + 1);

if (Lead2_turning_point)
{
    Lead2_turning_point_cnt++;
    if(Lead2_turning_point_cnt > 10)
    {
        Lead2_turning_point_cnt = 0;
        Lead2_turning_point = 0;
    }
}

if(Lead2_arrhythmia_show_cnt > -1)
{
    Lead2_arrhythmia_show_cnt++;
    if(Lead2_arrhythmia_show_cnt >= 400)
    {
        flg_4sec = TRUE;
        Lead2_arrhythmia_show_cnt = 0;
    }
}

*(bkWnd_Lead2 + bk_Pntr_Lead2) = Lead2_data ;

if (++bk_Pntr_Lead2 > 599) bk_Pntr_Lead2 = 0;

if (rrInt_Lead2 > 35)// pppp 35
{
    if ((Lead2_slp1 * Lead2_slp2) < 0)     // changed from 30
    {
        if (abs(Lead2_slp2) > 20)
        {
            if (abs(Lead2_slp3) > Lead2_qrs_peak * .25 )
            {
                if (((int)(abs(*(sldWnd_Lead2 + 5))) > Lead2_qrs_peak)
Lead2_qrs_peak = (int)abs(*(sld_wnd_Lead2 + 5));

// take absolute value

memcpy(Load2_R_peak_arr,
Load2_R_peak_arr + 1, 8 * sizeof(short));
*(Load2_R_peak_arr + 8) = abs(*(sld_wnd_Lead2 + 5));
++learn_count;
if(Lead2_qrs_peak > Lead2_R_Peak * 1.25 && learn_count >= 8)
{
learn_count = 8;
*(Load2_R_peak_arr + 8) = Lead2_R_Peak; // add the averaged r peak
if (peak_var == NV) // previous peak was abnormal
peak_wave_type = NV;
peak_var = NV;
}

else
{
peak_wave_type = NORMAL;
if (peak_var == NV) // previous peak was abnormal
peak_wave_type = NV;
peak_var = NORMAL;
}

Lead2_R_Peak = 0;
for (int j = 0; j < 9; j++)
{
    Lead2_R_Peak += *(Lead2_R_peak_arr + j);
}
Lead2_R_Peak = Lead2_R_Peak / 9;//this is the averaged r peak

//depending on the Lead2_R_peak calculate the
Lead2_turning_point = 0;
Lead2_turning_point_cnt = 0;
Lead2_check_for_width = TRUE;
if(peak_wave_type == NV
|| (rr_int_Lead2 < Lead2_average_R_R * .80 && Lead2_last_R_int != 0)
|| (rr_int_Lead2 > Lead2_average_R_R * 1.20 && Lead2_last_R_int != 0))
++err_counter;
if (rr_int_Lead2 < 45 && err_counter >= 1)
{
    //--err_counter; // checking for the noise // 27jan 2k3
    if(err_counter > 5 )
    {
        if (m_detectLead == V5)
        {
            m_detectLead = L2;
        }
    }
else
    {
        m_detectLead = V5;
    }
    CheckDetectionLead();
}
else
{
    if(err_counter > 5)// last five beats are having same pattern
    {
        // consider r to r stable rate
        err_counter = 0;
        learn_count = 0;
        Lead2_hr_sum -= *(Lead2_hr_array + Lead2_hr_array_cnt);
        *(Lead2_hr_array + Lead2_hr_array_cnt) = rr_int_Lead2;
        Lead2_hr_sum += *(Lead2_hr_array + Lead2_hr_array_cnt);
        Lead2_hr_array_cnt++;
        if(Lead2_hr_array_cnt >= AVG_CNT) Lead2_hr_array_cnt = 0;
        Lead2_last_avg_RR = Lead2_average_R_R;
        Lead2_average_R_R = Lead2_hr_sum / AVG_CNT;
        Lead2_last_R_int = rr_int_Lead2;
        if (Lead2_hr_sum > 800 && Lead2_hr_sum < 6400)
        {
            
        }
}
Lead2_heart_rate = (12000 * AVG_CNT)/ Lead2_hr_sum;
prev_HR_sum = Lead2_hr_sum;
}
}
else
{
Lead2_hr_sum -= *(Lead2_hr_array + Lead2_hr_array_cnt);
//VPC detected do the necessary operations
rr_int_Lead2 = Lead2_last_R_int;
*(Lead2_hr_array + Lead2_hr_array_cnt) = rr_int_Lead2;
Lead2_hr_sum += *(Lead2_hr_array + Lead2_hr_array_cnt);
Lead2_hr_array_cnt++;
if(Lead2_hr_array_cnt >= AVG_CNT) Lead2_hr_array_cnt = 0;
Lead2_last_avg_RR = Lead2_average_R_R;

Lead2_average_R_R = Lead2_hr_sum / AVG_CNT;
if (Lead2_hr_sum > 800 && Lead2_hr_sum < 6400)
Lead2_heart_rate = (12000 * AVG_CNT)/ Lead2_hr_sum; //pppp
}
}
else
{
// r to r stable rate
--err_counter;
if (err_counter < 0) err_counter = 0;
Lead2_hr_sum -= *(Lead2_hr_array + Lead2_hr_array_cnt);
//VPC detected do the necessary operations
*(Lead2_hr_array + Lead2_hr_array_cnt) = rr_int_Lead2;
Lead2_hr_sum += *(Lead2_hr_array + Lead2_hr_array_cnt);
Lead2_hr_array_cnt++;
if(Lead2_hr_array_cnt >= AVG_CNT) Lead2_hr_array_cnt = 0;

if (Lead2_average_R_R < 180 )
    Lead2_last_avg_RR = Lead2_average_R_R;
else
    Lead2_last_avg_RR = 180;
Lead2_average_R_R = Lead2_hr_sum / AVG_CNT;

Lead2_last_R_int = rr_int_Lead2;
if (Lead2_hr_sum > 800 && Lead2_hr_sum < 6400)
{
  Lead2_heart_rate = (12000 * AVG_CNT)/ Lead2_hr_sum;
  prev_HR_sum = Lead2_hr_sum;
}

Lead2_possible_RonT = FALSE;
if(Lead2_last_R_int < Lead2_last_avg_RR

3.11 Time Domain Analysis

The variations in heart rate may be evaluated by a number of methods. Perhaps the simplest to perform are the time domain measures. In these methods, either the heart rate at any point in time or the intervals between successive normal complexes are determined [48]. In a continuous ECG record, each QRS complex is detected, and the so-called normal-to-normal (NN) intervals (that is, all intervals between adjacent QRS complexes resulting from sinus node depolarization) or the instantaneous heart rate (IHR) is determined. Simple time domain variables that can be calculated includes the mean NN interval, the mean heart rate, the difference between the longest and shortest NN interval, the difference between night and day heart rate, and so forth. The simplest variable to calculate is the standard deviation of the NN intervals (SDNN), that is, the square root of variance. Since variance is mathematically equal to total power of spectral analysis; SDNN reflects all the cyclic components responsible for variability in the period of recording.
3.13 ECG Signal Display

Eight lead ECG signal is captured and converted to display 12 lead ECG output as per the relationships defined above. Signal is displayed on the ECG sheet defined in the form factor as per standard ECG sheet.

**Program Implemented**

```c
void CStPlotCtrl::PlotWave(short FAR* input_y)
{
    // Code using the ScreenDC & moveto lineto
    CDC *myDC;
    myDC = GetDC();
    Color cl(200,250,250,250);
    Graphics graphics(myDC->m_hDC);
    Pen t_pen(cl,2.0f);
    graphics.SetSmoothingMode(SmoothingModeAntiAlias);
    short vpb_value;
    int count = 0;

    int j;
    j = 0;
    int local_edge = dispArea.Width() - m_edge;
    // put the erase bar
    myDC->BitBlt(m_curPoint + 1, 0, 20,dispArea.Height(),&grid,
    m_curPoint + 1, 0, SRCCOPY);
    if (m_displayFormat == DISP_6_2)
    {
        myDC->BitBlt(m_curPoint + local_edge , 0,
    20,dispArea.Height(),&grid, m_curPoint + local_edge + 1, 0, SRCCOPY);
    }
    myDC->SelectObject(trace_pen);
}
```
if (m_displayFormat == DISP_6_2 || m_displayFormat == DISP_12_1)
{
    for (int i = 0; i < 6; i++)
    {
        CPoint curPoint(m_curPoint, *(m_offset + i) - *(input_y + i) * *(gain_arr + m_gain) / disp_scale);
        CPoint curPoint2(m_curPoint + local_edge, (int)(*(m_offset + i + 6) - *(input_y + i + 6) * *(gain_arr + m_gain) / disp_scale));
    
        /*
         * myDC->MoveTo(prevPoint[i]);
         * myDC->LineTo(curPoint);
         * myDC->MoveTo(prevPoint[i + 6]);
         * myDC->LineTo(curPoint2);
         */

        // for arrythmia detection
        if (i == 1)
        {
            analysis_buffer[count] = curPoint.y;
            count ++;

            if (count == 1200)
            {
                waveDetect.ResetMe();
                vpb_value = waveDetect.WaveDetect(analysis_buffer, x_location, y_location);
                count = 0;
                FireShowVPB(vpb_value);
            }
        }
    
}
graphics.DrawLine(&t_pen, prevPoint[i].x, prevPoint[i].y, curPoint.x, curPoint.y);
graphics.DrawLine(&t_pen, prevPoint[i + 6].x, prevPoint[i + 6].y, curPoint2.x, curPoint2.y);

if (flg_R)
{
    myDC->MoveTo(curPoint.x, dispArea.bottom - 5);
    myDC->LineTo(curPoint.x, dispArea.bottom - 15);
    flg_R = FALSE;
}
prevPoint[i] = curPoint;
prevPoint[i + 6] = curPoint2;
}
if (m_curPoint >= m_edge - 5)
{
    m_curPoint = 0;
    CRect rect;
    GetClientRect(&rect);

    for (int i = 0; i < 6; i++)
    {
        *(prevPoint + i) = 0;
        *(prevPoint + i + 6) = local_edge;
    }
    //m_prevValue = 0;
}
    m_curPoint++ ;
} else
{
}
//display format is selected three leads
for (int i = 0; i < 3; i++)
{
    switch(i)
    {
        case 0:
            j = m_displayLeads[0];
            break;
        case 1:
            j = m_displayLeads[1];
            break;
        case 2:
            j = m_displayLeads[2];
            break;
    }
    CPoint curPoint(m_curPoint, *(m_offset + i) -
    *(input_y + j) * *(gain_arr + m_gain) / disp_scale);
    myDC->MoveTo(prevPoint[i]);
    myDC->LineTo(curPoint);

    if (flg_R)
    {
        myDC->MoveTo(curPoint.x, dispArea.bottom - 5);
        myDC->LineTo(curPoint.x, dispArea.bottom - 15);
        flg_R = FALSE;
    }
    prevPoint[i] = curPoint;
}
if (m_curPoint >= m_edge - 10)
m_curPoint = 0;
CRect rect;
GetClientRect(&rect);

for (int i = 0; i < 6; i++)
{
    *(prevPoint + i) = 0;
}
m_prevValue = 0;
}
m_curPoint++;

if(flg_4sec)
{
    flg_4sec = FALSE;
}
ReleaseDC(myDC);
}

long CStPlotCtrl::GetCurPoint()
{
return m_curPoint;
}

void CStPlotCtrl::SetCurPoint(long nNewValue)
{
    m_curPoint = nNewValue;
    SetModifiedFlag();
}

void CStPlotCtrl::showGrid(const CRect *passRect)
{
    CDC *plotDC;
    const CRect *rect = passRect;
    int i;
// now calculate the offsets
setOffset(rect);
if ((plotDC = GetDC())== NULL)
{
    AfxMessageBox("Error Initialising Screen DC");
    exit(1);
}
// assign the pointer to screen DC
plotDC->SelectStockObject(DEFAULT_PALETTE);
plotDC->RealizePalette();

if (gridBMP.m_hObject != NULL)
    gridBMP.DeleteObject();

gridBMP.CreateCompatibleBitmap(plotDC,rect->Width(), rect->Height());
if (grid.m_hDC != NULL)
    grid.DeleteDC();
grid.CreateCompatibleDC(plotDC);

CBitmap *oldBMP = grid.SelectObject(&gridBMP);
// pass the DC & rectangle for drawing the grid
CBrush *brush;
// fill the background of the display
brush = new CBrush(m_backColor);
grid.FillRect(rect, brush);
delete brush;
DrawGrid(&grid, rect);

// put the lead labels
CFont *o_Font;
oldBMP = NULL;
grid.SetTextColor(m_textColor);
o_Font = grid.SelectObject(new_font);
char *lbl[] = { "I", "II", "III",
    "aVr", "aVL", "aVf",
    "V 1", "V 2", "V 3",
    "V 4", "V 5", "V 6"};

grid.SetBkMode(TRANSPARENT);
    if (m_displayFormat == DISP_12_1 || m_displayFormat ==
DISP_6_2)
    {
        for(i = 0; i < 6; i++)
        {
            grid.TextOut(rect->left, m_offset[i] + 5, lbl[i]);
            grid.TextOut(dispArea.Width() - m_edge, m_offset[i + 6] + 5,
lbl[i + 6]);
        }
    }
else
    {
        for(i = 0; i < 3; i++)
        {
            grid.TextOut(rect->left, m_offset[i] + 5,
lbl[m_displayLeads[i]]);
        }
    }

grid.SelectObject(o_Font);
// release the plotDC so there is no memory leak
grid.SetBkMode(OPAQUE);
ReleaseDC(plotDC);

}

void CStPlotCtrl::setOffset(const CRect *boundRect)
{

// calculate the offsets
int spc;

// display area of the control
dispArea.top = 0;
dispArea.bottom = boundRect->Height();
dispArea.left = 0;
dispArea.right = boundRect->Width();

int i;
switch (m_displayFormat)
{
    case DISP_6_2:
        spc = boundRect->Height() / 7;
        for (i = 0; i < 6; i++)
        {
            m_offset[i + 6] = m_offset[i] = spc + spc * i;
            prevPoint[i + 6] = m_offset[i + 6];
        }
        m_edge = boundRect->Width() / 2;
        sb_page_size = (long)(350);
        break;

    case DISP_12_1:
        spc = boundRect->Height() / 13;
        for (i = 0; i < 12; i++)
        {
            m_offset[i] = spc + spc * i;
            prevPoint[i] = spc + spc * i;
        }
        m_edge = dispArea.Width();
        sb_page_size = (long)(700);
        break;
}
case DISP_3_0:
    spc = boundRect->Height() / 4;
    for (i = 0; i < 3; i++)
    {
        m_offset[i] = spc + spc * i;
        prevPoint[i] = spc + spc * i;
    }

    m_edge = dispArea.Width();
    sb_page_size = (long)(700);
    break;
}

void CStPlotCtrl::OnTimer(UINT nIDEvent)
{
    int ch_no = 0;
    if (nIDEvent == 1)
    {
        // put the code for splitting the data and plotting the same
        while(comm.data_cnt > 1)
        {
            if (++rd_ptr >= 999) rd_ptr = 0;
            --(comm.data_cnt);
            // send one byte to the TM if present
            if(m_i2c && & TM_cnt)
            {
                CString tmp_st;
                tmp_st = TM_str[TM_str.GetLength() -
                TM_cnt];
                COleVariant var(tmp_st);
                comm.m_comm.SetOutput(var);
                --TM_cnt;
            }
        }
    }
}
COleControl::OnTimer(nIDEvent);
}

void CStPlotCtrl::StartTimer(long interval)
{
SetTimer( 1,interval, NULL);
}

void CStPlotCtrl::StopTimer()
{
KillTimer(1);
}

int CStPlotCtrl::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (COleControl::OnCreate(lpCreateStruct) == -1)
    return -1;
// Create the comm Dialog
comm.Create(IDD_DIALOG1);

// assign the pointer in the CComm class of this class
comm.StPlot = this;
comm.EnableAutomation();

new_font = new CFont;
    new_font->CreateFont(16,8,0,90,FW_BOLD,0,0,0,OEM_CHARSET,
                        OUT_DEFAULT_PRECIS,
                        CLIP_DEFAULT_PRECIS,
                        DEFAULT_QUALITY,
                        DEFAULT_PITCH, "Arial");

return 0;
}
short CStPlotCtrl::GetCommPort()
{
    return comm.m_comm.GetCommPort();
}

void CStPlotCtrl::SetCommPort(short nNewValue)
{
    ecg_port = nNewValue;

    try {

        comm.m_comm.SetCommPort(nNewValue);
    } catch (COleException * e) {
        throw (e);
    }

    SetModifiedFlag();
}

void CStPlotCtrl::StartComm()
{
    if(comm.m_comm.GetPortOpen() == FALSE)
        comm.m_comm.SetPortOpen(TRUE);
}

void CStPlotCtrl::StopComm()
{
    if(comm.m_comm.GetPortOpen() == TRUE)
        comm.m_comm.SetPortOpen(FALSE);
}
short CStPlotCtrl::GetInputLen()
{
return comm.m_comm.GetInputLen();
}

void CStPlotCtrl::SetInputLen(short nNewValue)
{
comm.m_comm.SetInputLen(nNewValue);
SetModifiedFlag();
}

short CStPlotCtrl::GetRefreshInterval()
{
return m_timer ;
}

void CStPlotCtrl::SetRefreshInterval(short nNewValue)
{
m_timer = nNewValue;
SetModifiedFlag();
}

void CStPlotCtrl::DoSetting()
{
if (!m_setting)
{
    comm.m_comm.SetInBufferCount(0);
    m_setting = TRUE;
    comm.wr_ptr = 0;
    comm.data_cnt = 0;
}
}
void CStPlotCtrl::PutData(unsigned char *data_val)
{
    short ecg_data[12], d_ecg_data[12];
    short unfiltered_data[8];
    short filtered_data[12];

    int i,j = 0;

    // get the 16 byte array from the comm port
    *(unfiltered_data + 0) =
    *(ecg_data + L2) = (long)(*(data_val + 0) * 4)
        + (*(data_val + 1) >> 6) - 410 ;

    *(unfiltered_data + 1) =
    *(ecg_data + L3) = (long)(*(data_val + 2) * 4)
        + (*(data_val + 3) >> 6) - 410 ;

    j = 2;
    for (i = 0; i < 6; i++)
    {
        j += 2;
        *(unfiltered_data + i + 2) =
        *(ecg_data + i + 6) = ((long)(*(data_val + j) * 4)
            + (*(data_val + 1 + j) >> 6) - 410 ;
    }

    *(ecg_data + L1) = *(ecg_data + L2) - *(ecg_data + L3) ;
    *(ecg_data + AVF) = (short)((ecg_data + L2) + *(ecg_data + L3)) / 1.732F ;
    *(ecg_data + AVL) = (short)((ecg_data + L1) + *(ecg_data + L3)) / 1.732F ;
*(ecg_data + AVR) = (short)(-(ecg_data + L1) + -(ecg_data + L2)) / 1.732F;

// save the unfiltered data
if (archive != NULL)
    SaveFile(ecg_data); // save to persistence

// put the filtering routing here
Notch(unfiltered_data, filtered_data);
LowPass(filtered_data, filtered_data);

// convert to 12 leads
*(ecg_data + L2) = *(filtered_data + 0);
*(ecg_data + L3) = *(filtered_data + 1);

for (i = 0; i < 6; i++)
    *(ecg_data + i + 6) = *(filtered_data + i + 2);

// recalculate with filtered data
*(ecg_data + L1) = *(ecg_data + L2) - *(ecg_data + L3);
    *(ecg_data + AVF) = (short)((ecg_data + L2) + *(ecg_data + L3)) / 1.732F;
*(ecg_data + AVL) = (short)((ecg_data + L1) + *(ecg_data + L3)) / 1.732F;
*(ecg_data + AVR) = (short)(-(ecg_data + L1) + -(ecg_data + L2)) / 1.732F;

DelayLowPass(ecg_data, d_ecg_data);
Spline(d_ecg_data, d_ecg_data);
RingFilter(d_ecg_data, d_ecg_data);

if(!auto_detect_flag)
    DetectHR(Rfilter(d_ecg_data[m_detectLead]),
              Rfilter(d_ecg_data[1]),
              Rfilter(d_ecg_data[7]), Rfilter(d_ecg_data[10]));
else

    d_ecg_data[10])),

    CLAMP(d_ecg_data[1], CLAMP_VOLTAGE, 200 ))

}